Science of Computer Programming 76 (2011) 555–586

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Metamodeling semantics of multiple inheritance

Roland Ducournau [a,*], Jean Privat [b]

[a] LIRMM – CNRS and Université Montpellier II, 161 rue Ada, 34000 Montpellier, France
[b] Dép. d'Informatique, UQAM, 210, avenue du Président-Kennedy, Montréal, QC, H2X 3Y7, Canada

ARTICLE INFO

Article history:
Received 26 December 2008
Received in revised form 5 October 2010
Accepted 21 October 2010
Available online 7 November 2010

Keywords:
Object-oriented programming
Multiple inheritance
Metamodeling
Redefinition
Linearization
Open-world assumption
Static typing
Virtual types

ABSTRACT

Inheritance provides object-oriented programming with much of its great reusability power. When inheritance is *single*, its specifications are simple and everybody roughly agrees on them. In contrast, *multiple inheritance* yields ambiguities that have prompted long-standing debates, and no two languages agree on its specifications. In this paper, we present a semantics of multiple inheritance based on metamodeling. A metamodel is proposed which distinguishes the "identity" of properties from their "values" or "implementations". It yields a clear separation between syntactic and semantic conflicts. The former can be solved in any language at the expense of a common syntactic construct, namely full name qualification. However, semantic conflicts require a programmer's decision, and the programming language must help the programmer to some extent. This paper surveys the approach based on *linearizations*, which has been studied in depth, and proposes some extensions. As it turns out that only static typing takes full advantage of the metamodel, the interaction between multiple inheritance and static typing is also considered, especially in the context of virtual types. The solutions proposed by the various languages with multiple inheritance are compared with the metamodel results. Throughout the paper, difficulties encountered under the *open-world assumption* are stressed.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

> Inheritance is commonly regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, but researchers rarely agree on its meaning and usage. Taivalsaari [96]

Class specialization and inheritance represent key features of object-oriented programming and modeling. Introduced in the Simula language [10], they have been related to the Aristotelian syllogistic [82–84] and contribute to the way the object-oriented approach meets software engineering requirements such as *reusability* and *extensibility*.

In spite of Taivalsaari's quotation above, inheritance is relatively simple when it is *single*, i.e. when a class cannot have more than one *direct superclass*—hence, the class specialization hierarchy is a tree or a forest. This is, however, a major limitation and there have been attempts since from the very beginning of object-oriented programming to soundly specify *multiple inheritance* in the pioneer object-oriented languages, i.e. Flavors [105], Smalltalk [12], and Simula [63]. It quickly appeared that multiple inheritance was not as simple as single inheritance since *conflicts* may occur that make the behavior hard to specify and give full meaning to the quotation. Different trends have divided the object-oriented programming community, with each one advocating a preferred policy.

* Corresponding author.
E-mail addresses: ducour@lirmm.fr, DUCOUR@LIRMM.FR (R. Ducournau), privat.jean@uqam.ca (J. Privat).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.10.006

1. A few production languages, but mainly Smalltalk [47], are in pure single inheritance, and their key feature is *dynamic typing*.
2. Several production languages, such as C++ [58,94], Eiffel [71,72], or Clos [92], propose full multiple inheritance. They were designed in the 1980s, and Python [101] is one of the few from the 1990s. All of them are widely used but have been the focus of considerable discussion, e.g. [6,91,19,104,86], and they all behave in a different way with respect to multiple inheritance.
3. Some languages are based on the close notions of *mixins* [13] or *traits* [27]. They are mostly research languages; Scala [78] is the most representative among recent ones, and Ruby [43] is one of the very few production languages that comply with this trend. A recent language, i.e. Fortress [2], is based on a close notion.
4. In the *static typing* setting, a major trend was inaugurated with Java *interfaces* [49], where classes are in single inheritance but with *multiple subtyping*, as a class can implement several unrelated interfaces; many recent languages, e.g. C# [73] and .Net languages, follow this trend.

Besides these programming languages, the main (or even only) modeling language, i.e. UML [80], includes multiple inheritance without any precise specification.

The need for multiple inheritance has also prompted a long-standing debate. Besides the aforementioned references, see for instance [88] and various related conference panel sessions. However, the fact that few statically typed languages use pure single inheritance, i.e. *single subtyping*, strongly underlines the importance of multiple inheritance. The rare counterexamples, such as Oberon [106,75], Modula-3 [50], or Ada 95 [8], result from the evolution of non-object-oriented languages. Furthermore, the absence of Java-like multiple inheritance of interfaces was viewed as a deficiency of the Ada 95 revision, and this feature was incorporated in the next version [95]. The requirement for multiple inheritance is less urgent in the dynamic typing framework; for instance, all Java multiple subtyping hierarchies can be directly defined in Smalltalk, by simply dropping all interfaces. Conversely, statically typing a Smalltalk hierarchy only involves adding new interfaces to *introduce* methods that are *introduced* by more than one Smalltalk class.[1]

Overall, despite the numerous dedicated works, multiple inheritance is not a closed issue. From this standpoint, there is no satisfactory language. As we shall see, even languages based on multiple subtyping may be flawed, since they may not ensure full reusability. In this paper, we propose a semantics of class specialization and inheritance which is "natural", or even "Aristotelian". Our proposal aims to ensure universality and simplicity by (i) clarifying concepts and clearly defining problems related to multiple inheritance in general; (ii) identifying specific multiple inheritance issues that are usually unresolved (or poorly resolved) and proposing solutions for them; and (iii) formally discussing and comparing the different specifications of many OO languages (C++, Java, Clos, Python, C#, etc.).

The proposed semantics is based on *metamodeling*, i.e. reifying the concerned entities, namely classes and properties. The proposed metamodel is the simplest metamodel that models classes and properties in such a way that each name in the program code can denote a single instance of the metamodel. It allows us to get rid of names and their associated ambiguities in order to just consider reified entities. In contrast, most object-oriented languages, especially in static typing, attempt to interpret names and inheritance in the Algol tradition, on the basis of the scope and extent—e.g. the so-called *scope resolution operator* in C++. Whereas it can work with single inheritance, i.e. the subclass is interpreted as a block nested in its superclass, it obviously fails with multiple inheritance. The first benefit drawn from this metamodel is to precisely distinguish the "identity" of a property from its "value" (or "implementation") in a given class. In turn, it strongly distinguishes between two kinds of conflict which should not be confused, even though they are currently confused in all known languages with multiple inheritance. The first kind of conflict involves property names, and occurs when a class inherits two properties with the same name or signature, which have however been introduced in unrelated classes. The second kind of conflict occurs when a class cannot choose between different implementations for the same property. A variant concerns the case where several implementations must be combined. These two categories are quite different and require different answers. The first kind is purely syntactical, and a simple unambiguous denotation would provide a solution; for instance, it would make Java fully reusable in the sense that any pair of unrelated class/interface could be specialized by a common subclass. The second kind of conflict involves the program semantics; the solution cannot rely on some syntactic feature but the languages should offer the programmer some help for managing them. One approach has been studied in depth, namely *linearization* [32,33,55,34–36,9,40,45]. These two inheritance levels were originally identified in [36], but the lack of a metamodel hindered the authors from finishing the analysis.

The approach proposed in this paper would apply to all object-oriented programming languages with multiple inheritance, multiple subtyping, or even mixins. However, it turns out that *static typing* is required to take full advantage of the metamodel. Therefore the paper is focused on statically typed languages. Without loss of generality, this proposal cross-cuts usual type theories and object calculi. Usual type theories, e.g. record types [16], are based on names, and substituting reified properties for names does not change the considered type theory. However, multiple inheritance conflicts and some related solutions have special effects on types when redefinition is not type invariant. Hence, the metamodel is also applied to *virtual types* [98], and this paper analyzes the way static typing and multiple inheritance interact.

[1] The "introduction" term is crucial here and will be more formally defined. A class *introduces* a method when it defines a method with a *new* name (or signature) that is not already defined in any of its superclasses.

Overall, a major problem is addressed throughout the paper, namely whether the *open-world assumption* (OWA) holds or not. It concerns both the design level and run-time systems. We consider that the object-oriented philosophy is best expressed under the OWA. Each class must be designed and implemented while ignoring how it will be reused, especially whether it will be specialized in single or multiple inheritance. This is the essence of reusability, not only a symptom of claustrophobia [6]. However, under the *closed-world assumption* (CWA), when the entire class hierarchy is known, detecting and fixing conflicts is easier, and implementation can be as efficient as with single inheritance. In contrast, under the OWA, e.g. in a dynamic loading setting, it is not possible to foresee conflicts which might occur when defining or loading future classes, and this may affect the implementation efficiency.

The article is organized as follows. Section 2 presents metamodeling, describes the proposed metamodel, i.e. a UML model of classes and properties, and analyzes the meaning of class specialization and property inheritance. The metamodel is formalized in a simple set-theoretical way. Section 3 examines both kinds of conflict and different ways of solving them. Section 4 reviews the main results on the linearization approach and proposes some new extensions. Section 5 examines multiple inheritance from a static typing standpoint, with non-invariant parameter and return types. Method combination, virtual types, and parametric classes are also considered. In Section 6, specifications of the most commonly used object-oriented languages are compared with the metamodel, and the *mixin* alternative is examined. Finally, the last section concludes the paper by presenting current prospects along with the known limitations of our proposal.

2. Class and property metamodel

In this section, we first informally present the key notions of specialization and inheritance by coming back to the Aristotelian syllogistic. Then we discuss the metamodel notion, especially in an object-oriented framework, and we propose a UML metamodel with its formalization in a simple set-theoretic setting.

2.1. Specialization and inheritance

A *de facto* standard object model is a *class-based* model, as opposed to other approaches like *actors* or *prototypes* [70]. It consists mainly of three kinds of entity: (i) *classes* are organized in a *specialization* hierarchy; (ii) *objects* are created as instances of these classes by an instantiation process; and (iii) a set of *properties* describes each class, with *attributes* for the state of its instances and *methods* for their behavior. Moreover, a property defined in a class may be *redefined* (or *overridden*) in a subclass. Finally, applying a method to an object, called the *receiver*, follows the metaphor of *message sending* (also called *late binding* or *dynamic binding*). The invoked method is selected at run time according to the class of the receiver. This is the core of the model, and discussing the specialization semantics does not need more detailed specifications.

Though a novel feature in computer science, specialization has quite ancient roots in the Aristotelian tradition, for instance in the well-known syllogism: *Socrates is a human, humans are mortals, thus Socrates is a mortal*. Here, *Socrates* is an instance, while *human* and *mortal* are classes. Interested readers will find an in-depth analysis of the relationships between object orientation and Aristotelian syllogistic in [83,84]. Here, we use "Aristotelian" as an anchor to this bimillenial tradition which is the basis of our modern common-sense understanding of the real world. According to the Aristotelian tradition, as revised with computer science vocabulary, one can generalize this example by saying that *instances of a class are also instances of its superclasses*. More formally, $\prec$ is the specialization relationship ($B \prec A$ means that B is a subclass of A) and *Ext* is a function which maps classes to sets of their instances, i.e. their *extensions*. Then:

$$B \prec A \Longrightarrow Ext(B) \subseteq Ext(A). \tag{1}$$

This is the essence of specialization, and it has a logical consequence, namely the inclusion of intensions, i.e. inheritance. When considering the properties of a class, one must remember that they are properties of instances of the class and are factorized in the class. Let B be a subclass of A; then, with instances of B being instances of A, they have all the properties of instances of A. One says that subclasses inherit properties from superclasses. More formally, *Int* is a function which maps classes to sets of their properties, i.e. their *intensions*. Then:

$$B \prec A \Longrightarrow Int(A) \subseteq Int(B). \tag{2}$$

In the Aristotelian terminology, B is defined by *genus* (i.e. A) and *differentia*, i.e. $Int(B) \setminus Int(A)$.

The intuition of what are specialization and inheritance is well captured by the Aristotelian syllogistic. However, key points remain to be examined. What is a property? What are the relationships between classes and properties? Moreover, property redefinition and late binding are not covered by Aristotelian syllogistics.

2.2. Models and metamodels

Modeling is certainly one of the most fundamental activities of scientists. A variety of technical means can be employed, from drawings to mathematics. Anyway, in software engineering, the standard is now object-oriented modeling, which is best represented by UML, the *Unified Modeling Language* [80]. UML and object-oriented modeling are now of common practice. Therefore, we just recall the main notions, especially for readers who are not familiar with metamodels.

2.2.1. From modeling to object-oriented metamodeling

Here we will overlook the graphical aspects of UML, and only consider entities that underlie class and instance diagrams. Roughly speaking, apart from the procedural code that is not considered, an object-oriented model is close to an object-oriented program. Both involve classes and properties, i.e. attributes and methods. Models add a specific notion called *association* that represents relations between classes. At the instance level, a model consists of a graph, in the graph-theoretic sense, of objects that are related by associations described at the class level. As we do not consider the dynamic aspects of UML, such a model is static. Though UML does not enforce a precise formal semantics, we associate with such a model an intuitive set-theoretic interpretation: classes are extensions, i.e. sets of objects, associations are binary relations between these sets, and specialization is the Aristotelian inclusion of extensions.

Modeling, especially when object-oriented, proceeds by *reification* and *abstraction*. Through reification, an abstract entity is changed into a concrete object. In contrast, abstraction involves jumping from the subclass to the superclass, i.e. generalization, or from the object to the class level, i.e. *meta-abstraction*. *Meta* is a key notion in mathematics and computer science. For instance, logic (e.g. [60]) makes an essential distinction between the *language* of formal logic, e.g. first-order logic (FOL), and *metalanguages* that allow us to speak about FOL phrases in formal or informal ways. Inference rules, e.g. *modus ponens*, thus represent a metalanguage that gives a formal semantics to FOL. An analogous approach can be followed with modeling. A *metamodel* is an explicit model of constructs and rules that are needed to build specific models within a domain of interest. Any kind of modeling methodology can be used for metamodeling. However, we shall restrict the range of metamodeling in two ways. First, we only consider metamodels that are object-oriented models. Second, our metamodel will focus on the core of object-oriented notions, such as classes, properties, specialization, and inheritance. Hence, the metamodels considered can be thought of as *reflective object-oriented metamodels* or, alternatively, *object-oriented models of object-oriented languages*.

Overall, modeling is an intuitive way of considering the modeled reality, here an object-oriented program, that provides a semi-formal semantics. In this setting, metamodeling supports intuition and serves as an operational semantics when some meta-object protocol is added. A nice example is the analysis of classes, metaclasses, and instantiation made in the ObjVLisp model [24]. Although the reflective kernel of ObjVLisp is also present in other proposals for reflection, e.g. [59,45], we take it as an example because it focuses on its object, namely classes and metaclasses in their purest form. Admittedly, UML is designed as a stack of metamodels through the MOF (*Meta-Object Facility*). However, these metamodels are not useful for our goal of formalizing multiple inheritance, and we shall propose our own original metamodel. Metamodeling some part of an object-oriented programming language thus amounts to defining an object model (i.e. entities like classes, associations, attributes, methods, etc.) for modeling the language concepts considered. We also consider that such a metamodel should specify the meaning of names in programs in such a way that all names should contextually be unambiguous. This is expressed by the following requirement.

Requirement 2.1 (*Mapping from Syntax to Model*)**.** *In the modeled program, any occurrence of an identifier denoting a modeled entity must unambiguously map to a single instance of the metamodel. In other words, names must provide a mapping from syntax to model.*

Accordingly, metamodeling allows us to get rid of names when considering the modeled entities. This will prove to be of great value, as most difficulties yielded by object-oriented programming lie in the interpretation of names. Besides the formalization of basic object-oriented concepts which is covered in the following sections, the expected advantage of such a metamodel is that all programming tools could just consider reified entities instead of names. These reified entities actually represent the reality (*ontology*) of object-oriented programs, and names should remain at the human–computer interface.

Conversely, according to Occam's Razor, metamodeling should aim at minimality. As a counterexample, the Clos reflective kernel includes, but is far larger than, the ObjVLisp model. It may be necessary for fully implementing Clos, but it is useless for conceptually modeling classes, superclasses, and instantiation. Models must remain partial—this is of course a truism, since a complete model is the real world itself.

2.3. The UML *model of classes and properties*

We now present a metamodel for classes and properties in object-oriented languages. Here we only consider properties that: (i) are described in a class but dedicated to its instances; and (ii) depend on the *dynamic type* (also known as the *run-time type*) of the object considered, i.e. the class which has instantiated it. They could be qualified as *virtual* in C++ jargon, or even tagged by the `virtual` keyword in the method case. "Virtual" comes from Simula and has several usages: virtual functions (Simula, C++, C#); virtual types and classes (Simula, Beta [67,68]); and virtual multiple inheritance (C++). It can be understood as "redefinable", in the sense of "redefinition" in the present paper, and hence submitted to *late binding* and depending on the dynamic type of some *receiver*. However, "virtual" has also been used in the sense of "abstract" (also known as *deferred*), i.e. for a non-implemented method or a non-instantiable class, e.g. in [81,56] and in Ocaml [51]. Though both meanings are related ("abstract" implies that something is "virtual"), they are different enough to require different terms, and we only consider the former usage. Class properties, i.e. properties which only concern the class itself, and not its instances, are excluded from the scope of the metamodel. For instance, the fact that a class may be *abstract* is not considered as a property. We do not consider *static* methods or variables either. In Clos, however, we may consider slots declared with `:allocation :class` because they are dedicated to instances though shared by all of them.

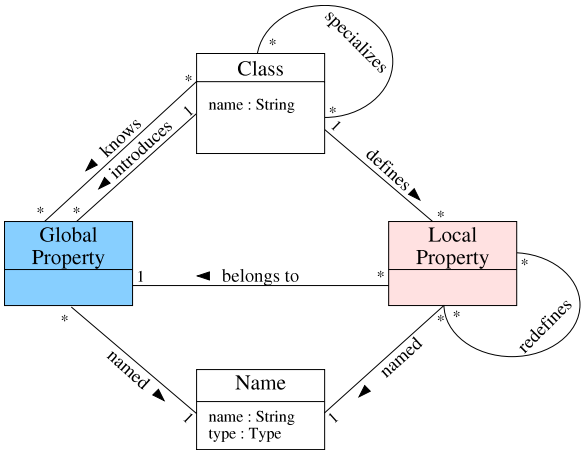


Fig. 1. Metamodel of classes and properties.

This metamodel is intended to be both intuitive and universal. It is likely in line with the intuition of most programmers with respect to object-oriented concepts. It is universal in the sense that it is not dedicated to a specific language, and it is very close to the specifications of most statically typed object-oriented languages, at least when they are used in a simple way. Hence, it can be considered as an implicit metamodel of most languages, even though no language strictly complies with it. However, this metamodel has never been explicitly described in any programming language or even in UML. In the following, we successively present a UML model which provides an informal idea of the metamodel, a small example, then a more formal set-theoretical definition. The section ends by considering the resulting run-time behavior. The analysis of multiple inheritance conflicts will be tackled in Section 3.

2.3.1. Classes and properties

The metamodel consists of three main kinds of entity: *classes*, *global properties*,[2] and *local properties* (Fig. 1). The former is natural, but the latter two are original key features of the model. It follows from Requirement 2.1 that late binding (also known as message sending) implies the definition of exactly two property categories. *Local properties* correspond to attributes and methods as they are defined in a *class*, independently of other possible definitions of the "same property" in superclasses or subclasses. *Global properties* are intended to model this "same property" idea in different related classes. They correspond to messages that the instances of a class can answer. In the case of methods, the answer is the invocation of the corresponding local property of the dynamic type of the receiver. Each local property *belongs to* a single global property and is *defined* in a single class. Finally, a last kind of entity represents names which are crucial and can be less simple than usual symbols. Names can currently be considered as simple symbols.

The associations in Fig. 1 represent verbs that can be used for relating a property to a class. Throughout the article, they will be used in this formal way. However, we shall also follow a common linguistic usage, namely metonymy. For instance, formally, a class defines local properties, and stating that it defines a global one means that the class defines a local property belonging to the global one.

Global and local properties must in turn be divided into several specific kinds according to the associated values: data for attributes or functions for methods. Attributes and methods are present in all languages, but the main complication involves the methods. Indeed, attributes are usually quite simpler, though languages such as Clos or Eiffel accept the full attribute redefinition. Moreover, a proper distinction between attributes and methods may not be straightforward since it is relevant to accept, as in Eiffel, that a functional method without parameters can be redefined as an attribute. However, there is no need to detail this here. Besides attributes and methods, a third kind of property must also be considered, namely *virtual types*. Whereas attributes and methods are respectively associated with data and functions, virtual types involve associating a property with a type that depends on the dynamic type of the receiver. Virtual types [98,97,56] represent a combined genericity and covariance mechanism first introduced in Beta [67,68], and they are somewhat similar to Eiffel *anchored types* [72]. The fact that functions and types are part of the metamodel does not imply that they must be first-class objects in the considered programming languages. Other properties, such as Eiffel *class invariants*, can be reduced to these three basic ones. Anyway, in the following, "property" stands for all three kinds and a partial but more intuitive translation of our terminology is possible; global (respectively, local) properties stand for methods (respectively, method implementations). In this section, the specific kind of property does not matter, and the case of virtual types will be examined in Section 5.

[2] In some preliminary papers, the term "generic property" was used instead of "global property". It was coined on the basis of Clos *generic functions*, which is closely related, apart from method dispatch. Zibin and Gil [107] use the term "method family" or "implementation family" in a close though informal sense. "Genus" is of course the Latin word for "family", and "generic" would seem to be more appropriate than "global". However, because of the possible confusion with so-called *generics*, i.e. *parameterized polymorphism* that applies to types, classes or functions, we finally prefer "global".

Classes are organized in a hierarchy, actually a *dag* (directed acyclic graph), by the *specialization* relation. A class definition is a triplet consisting of a class name, the name of its superclasses (presumably already defined), and a set of local property definitions. As already mentioned, the specialization relation supports the inheritance mechanism, i.e. classes inherit the properties of their superclasses. When translated in terms of the metamodel, this yields two-level inheritance. First of all, the new class *inherits* all global properties from its superclasses—this is *global property inheritance*. The class *knows* all the global properties *known* by its superclasses. Then each local property definition is processed. If its name is the same as that of an inherited global property, the new local property is attached to the global property. This implicitly defines the *redefinition* relation between the new local property and the inherited ones. Otherwise, if there is no such inherited global property, a new global property with the same name is *introduced* in the class.

Local property inheritance takes place at run time, though implementations can statically precompute it. A call site `x.foo(args)` represents the invocation of the *global property* named `foo` of the *static type* (say *A*) of the receiver `x`. The static typing requirement appears here, and in a dynamic typing framework, for instance in Smalltalk, there is no way to distinguish between different global properties with the same name while meeting Requirement 2.1. Anyway, at run time, this call site is interpreted as the invocation of the **single** *local property* corresponding to both the *global property*, and the *dynamic type* of the value bound to `x`, i.e. the class which has instantiated it. Therefore, when no such local property is defined in the considered class, a local property of the same global property must be inherited from superclasses. Static typing ensures that both global and local properties exist.

Static overloading. In many statically typed languages, such as C++, Java, or C#, a class may know several properties with the same name and different parameter types and numbers. This slightly complicates naming. The property names in the language and in the metamodel must be distinguished. In the language, the names can be overloaded but, in the metamodel, they are made locally unambiguous by considering the tuple formed by the name and the parameter types. This is known as *name mangling* in compiler jargon, and is the reason for reifying property names. Hence, names are signatures, i.e. the symbol plus the parameter types, and two overloaded properties correspond to different global properties. Let us consider the `x.foo(args)` call site when `foo` is overloaded in *A* (the static type of `x`), i.e. when *A* knows several global properties named `foo`, which differ by their parameter types or number. Then a **single** global property, i.e. the most specific according to formal and actual parameter types, must first be selected *at compile time*. A conflict may occur when the most specific global property is not unique, e.g. when there is multiple inheritance between parameter types, or with multiple contravarying parameters. Such conflicts can be easily solved by simply making actual parameter types more precise at the call site. Paradoxically, in this context, "more precise" means more general, since only supertypes can disambiguate. *At run time*, the call site remains interpreted as the invocation of the **single** *local property* corresponding to this **single** statically selected *global property* and to the *dynamic type* of the value bound to `x`. Overall, static overloading is error prone, because static and dynamic selections are hard for programmers to distinguish, and thus it is often confused with *covariant redefinition* (Section 5) or *multiple dispatch* (Section 6.3) [53].

Static properties. We now further discuss our exclusion of all static properties. The original object-oriented philosophy states that the program behavior is determined by the object itself. In a class-based language, all instances of the same class have the same behavior, which is thus determined by the object dynamic type. In contrast, a static method (in Java) or a non-virtual function (in C++) are determined at compile time. Only non-virtual functions imply a `this` parameter. It would be possible to integrate static or non-virtual methods in the metamodel by simply considering that they represent global and local properties in a one-to-one correspondence, instead of the general one-to-many case. This is, however, useless because static properties only consider classes as name-spaces. Hence, if static method `foo` is defined in class `A`, a call to `A.foo` is exactly equivalent to, say, the static function `A_foo` in a procedural language. Admittedly, a form of inheritance seems possible since, if `B` is a subclass of `A`, `B.foo` is equivalent to `A.foo`. However, as everything here is resolved statically, i.e. at compile time, possible ambiguities resulting from multiple inheritance or static overloading can be solved by the programmer without any need for special constructs. The case of attributes is similar, although deciding whether attributes must be considered virtual or not is a rather Byzantine question. In Eiffel, they are because their type can be redefined. In C++, Java, and C#, they cannot be redefined, and they might be considered as non-virtual. However, we consider that they are virtual (i) because they are specific to each object, (ii) because their position in the object layout depends on the object dynamic type, and (iii) for the sake of uniformity, since they are redefinable in some languages. In contrast, static variables could just be considered as shared static attributes. Overall, there is theoretically no need to include these static properties in the metamodel, and in practice including them would be trivial.

Constructors. Constructor methods are special two-sided functions. As a local property, a constructor is just an *initializer*, which corresponds to Clos `initialize-instance`. At a constructor call site, i.e. as a global property, the construction role is fulfilled by some hidden mechanism similar to Clos `make-instance`, which makes the instance before the initializer is called. Therefore, one might imagine that an object-oriented language only needs a special operator (say `new`) for instantiating a class, and then any method could be applied as an initializer. This would, however, be a poor way to deal with uninitialized attributes, especially for references. All languages thus specify constructors in a rather restrictive way. In C++, Java, and C#, they are static methods. Moreover, in C++, late binding does not apply inside a constructor because of the lack of `null`-initialization in this language. In spite of this restriction, in C++, uninitialized attributes are still a source of fatal error like *segmentation faults*. In contrast, Eiffel constructors are ordinary methods, which can be redefined, but their

```
1   class A {
2       public void foo()
        {...}
    }

3   class B extends A {
4       public void foo()
        {...}
5       public void bar()
        {...}
    }

    ...

    {...
1    A x
3    B y
6    x.foo()
7    y.bar()
     ...}
```

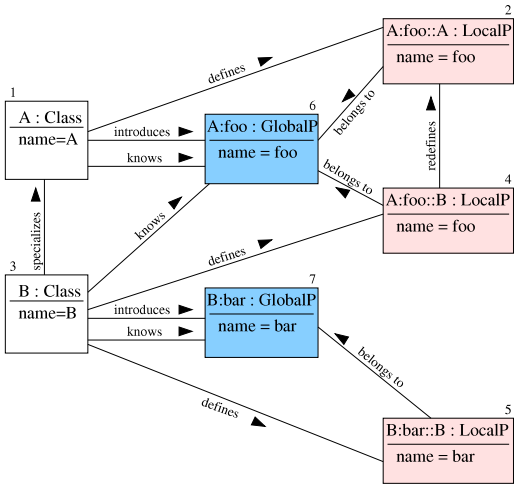


Fig. 2. A simple Java example and the corresponding instance diagram. For simplicity, the names are no longer reified.

construction role is not inheritable. Constructors do not need to be included in the metamodel since they reduce to more ordinary properties.

2.3.2. Example

The Java example in Fig. 2 defines seven entities of our metamodel: two classes, `A` and `B`; three local properties, the `foo` method defined in `A` and the `foo` and `bar` methods defined in `B`; and two global properties, respectively introduced as `foo` in `A` and `bar` in `B`. The corresponding numbered instance diagram specifies the unambiguous mapping between the code and the instances of the metamodel (Requirement 2.1).

The instantiation of the metamodel proceeds as follows, as the code is read:

- class `A` is first created (1); it does not inherit any explicit global property; in practice, it would inherit all global properties introduced in the hierarchy root, `Object`;
- the `foo` method is defined in `A`: the corresponding local property is created (2) and, since `A` does not know any global property with this name, a `foo` global property is introduced (6);
- class `B` is then created (3) as a specialization of `A`; it then inherits all explicit global properties from `A`, especially the `foo` global property;
- the `foo` method is defined in `B`: the corresponding local property is created (4) and attached to the `foo` global property (6) inherited from `A`—this is a redefinition;
- the `bar` method is defined: the corresponding local property is created (5) and, since `B` does not know any `bar` global property, then a `bar` global property is introduced (7);
- in the following code sequence, `foo` (respectively, `bar`) is understood as being the single `foo` (respectively, `bar`) global property which is known by the static type `A` (respectively, `B`) of the receiver `x` (respectively, `y`); changing the static type of `x` from `A` to `B` would not change the mapping;
- finally, at run time, the invocation of `foo` will call the local property defined in `A` or in `B`, according to the actual dynamic type of the value of `x`—this is late binding, as usual.

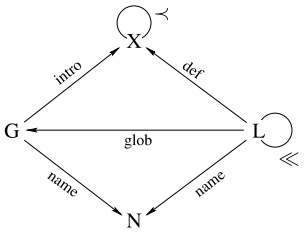
In the diagram of Fig. 2, there are two forms for the names of all considered instances. A global property is denoted with the "`:`" operator, for instance by `A:foo`, where `A` is the name of the introduction class and `foo` is the property name. A local property is denoted with the "`::`" operator, for instance `A:foo::B`, where `A:foo` denotes the global property and `B` is the name of the definition class. This is metalanguage. The figure clearly suggests how a development tool like Eclipse [46] could allow a programmer to easily navigate between the source code and the model instances.

2.4. Formal definitions

The rest of Section 2 formalizes this first informal presentation. Readers mainly interested in semantic discussions might want to skip it and jump directly to Section 3. However, all semantic definitions rely on the notations presented here. These notations are also recalled in Table 1. We first define a model in a static way, i.e. its components and their relationships, then describe the protocols: (i) for instantiating it, and (ii) for late binding.

Table 1
Index of notations and functional view of the metamodel.

Sets			Page
X		Classes	562
G		Global properties	562
L		Local properties	562
N		Names	562
Relations			
$\prec$	$X \times X$	Specialization	562
$\ll$	$L \times L$	Redefinition	562
$\leq_{clin(c)}$	$supc(c) \times supc(c)$	Class linearization	571
$\leq_{llin(g,c)}$	$loc(g,c) \times loc(g,c)$	Property linearization	571
Functions			
def	$L \to X$	Definition class	562
glob	$L \to G$	Global property	562
intro	$G \to X$	Introduction class	562
name	$G \uplus L \to N$	Property name	562
gid	$G \to X \times N$	Global property identifier	563
lid	$L \to G \times X$	Local property identifier	564
loc	$G \times X \to 2^L$	Local properties	565
sel	$G \times X \to L$	Selected local property	565
spec	$G \times X \to 2^L$	Most specific local properties	565
cs	$G \times X \to 2^X$	Class conflict set	565
supl	$L \to 2^L$	Masked local properties	566
supc	$X \to 2^X$	Superclasses	570
clin	$X \to (X \to \mathbf{N})$	Class linearization	571
llin	$G \times X \to (L \to \mathbf{N})$	Property linearization	571
cnm	$X \times L \to L$	Next method	571
lpo	$X \to (X \to \mathbf{N})$	Local precedence order	572



Notations

Let E, F, and G be sets. 2^E denotes the power set of E, $|E|$ is the cardinality of E, and $E \uplus F$ is the union of the disjoint sets E and F. Given a function $foo : E \to F$, the function $foo^{-1} : F \to 2^E$ maps $x \in F$ to the set $\{y \mid foo(y) = x\}$. Function notations are extended to powersets and Cartesian products in the usual way: $\forall G \subseteq E, foo(G) = \{foo(x) \mid x \in G\}$ and $\forall R \subseteq E \times E, foo(R) = \{(foo(x), foo(y)) \mid (x, y) \in R\}$. Finally, $(E, \leq)$ denotes the graph of a binary relation $\leq$ on a set E. It is a *poset* (partially ordered set) iff $\leq$ is reflexive, transitive and antisymmetric. Then, $\max_{\leq}$ (respectively, $\min_{\leq}$) denotes the *maximal* (respectively, *minimal*) elements of a subset of E according to $\leq$.

Definition 2.2 (*Class Hierarchy*)**.** A *model of a hierarchy* $\mathcal{H}$, i.e. an instance of the metamodel, is a tuple $\langle X^{\mathcal{H}}, \prec^{\mathcal{H}}, G^{\mathcal{H}}, L^{\mathcal{H}}, N^{\mathcal{H}}, name_{\mathcal{H}}, glob_{\mathcal{H}}, intro_{\mathcal{H}}, def_{\mathcal{H}} \rangle$, where

- $X^{\mathcal{H}}$ is the set of *classes*;
- $\prec^{\mathcal{H}}$ is the *class specialization* relationship, which is transitive, antisymmetric and anti-reflexive; $\preceq^{\mathcal{H}}$ (respectively, $\prec_d^{\mathcal{H}}$) denotes the reflexive closure (respectively, transitive reduction) of $\prec^{\mathcal{H}}$ and $(X^{\mathcal{H}}, \preceq^{\mathcal{H}})$ is a poset;
- $G^{\mathcal{H}}$ and $L^{\mathcal{H}}$ are disjoint sets of *global* and *local properties*;
- $N^{\mathcal{H}}$ is the set of *identifiers* (names) of *classes* and *properties*;
- $name_{\mathcal{H}} : G^{\mathcal{H}} \uplus L^{\mathcal{H}} \to N^{\mathcal{H}}$ is the naming function of properties;
- $glob_{\mathcal{H}} : L^{\mathcal{H}} \to G^{\mathcal{H}}$ associates a global property with each local property;
- $intro_{\mathcal{H}} : G^{\mathcal{H}} \to X^{\mathcal{H}}$ associates an introduction class with a global property;
- $def_{\mathcal{H}} : L^{\mathcal{H}} \to X^{\mathcal{H}}$ associates with a local property the class in which it is defined.

Sets $X^{\mathcal{H}}$, $G^{\mathcal{H}}$, and $L^{\mathcal{H}}$ correspond to the three classes in the metamodel upper triangle, whereas the total functions $glob_{\mathcal{H}}$, $intro_{\mathcal{H}}$, and $def_{\mathcal{H}}$ correspond to the three functional associations. The "specializes" association is represented by $\prec^{\mathcal{H}}$ and all other associations, such as "knows" and "redefines", are not primitive. On the other hand, $N^{\mathcal{H}}$ and $name_{\mathcal{H}}$ represent relationships between the metamodel and names which are used in the program text. These names can be simple symbols, or more complex objects if static overloading is considered. In this situation, method names must include a symbol and the parameter types which are class names. Indeed, classes, too, are named, but this does not have to be explicit in the metamodel. So far, the formalization is a straightforward translation of the UML diagram in Fig. 1. The notations are supplemented by the following set of equations and definitions (3)–(11). The metamodel is generic, as all its components are parameterized by $\mathcal{H}$. However, in the rest of the paper, parameter $\mathcal{H}$ will remain implicit for the sake of readability. The parameter must be explicit when several hierarchies are considered, e.g. as in [37].

A *legal model* will be defined by a set of constraints which ensure that (i) the two triangular diagrams commute at the instance level; and (ii) names in the program text are unambiguous or can be disambiguated. There are several ways to do this, and we shall follow a prescriptive approach in the rest of Section 2 by presenting what we think is the best choice. In return, we shall also give some hints for alternatives and discuss the choices. The set-theoretic formalization that follows does not involve non-trivial properties; hence it does not require formal proofs.

2.4.1. Global properties

Given a class $c \in X$, G_c denotes the set of global properties *known* by c. Global properties are either *inherited* from a superclass of c, or *introduced* by c. Let $G_{\uparrow c}$ and G_{+c} be the two corresponding subsets. Hence, all G_{+c} are disjoint with each other, and

$$G_{+c} \triangleq intro^{-1}(c), \tag{3}$$

$$G_{\uparrow c} \triangleq \bigcup_{c \prec_d c'} G_{c'} = \biguplus_{c \prec c'} G_{+c'}, \tag{4}$$

$$G_c \triangleq G_{\uparrow c} \uplus G_{+c} = \biguplus_{c \preceq c'} G_{+c'}, \tag{5}$$

$$G = \bigcup_{c \in X} G_c = \biguplus_{c \in X} G_{+c}. \tag{6}$$

Formulas (3)–(5) formally define *global property inheritance*.

The point with global properties is mostly a matter of names. The metamodel must therefore specify how a name can be used in a particular context and which conflicts can result from a given model.

Constraint 2.3 (*Locally Unambiguous Names*)**.** *When a global property is introduced, its identifier must be unambiguous; hence, for all* $c \in X$, (i) *the restriction of name to* G_{+c} *is injective and* (ii) *inherited and introduced properties cannot have the same name:*

$$name(G_{+c}) \cap name(G_{\uparrow c}) = \emptyset.$$

This constraint is actually implied by the local property constraints (Section 2.4.2). We thus do not consider here the questionable possibility of introducing a new global property with the same name as an inherited one, as with the `reintroduce` and `new` keywords, respectively, in Object Pascal and C#. This form of static overloading would pose similar problems and require similar solutions (Section 2.3.1). The constraint also implies that the function $gid : G \to X \times N$ that maps a global property $g \in G$ to the pair

$$gid(g) \triangleq (intro(g), name(g)) \tag{7}$$

is injective. The syntax `A:foo` used in Section 2.3.2 represents *gid*. When overlooking the local properties, all $\langle X, \prec, G, N, name, intro\rangle$ tuples that satisfy Definition 2.2 and Constraint 2.3 are legal.

Definition 2.4 (*Global Property Conflict*)**.** Given a class $c \in X$ and two distinct global properties $g_1, g_2 \in G_{\uparrow c}$, a *global property conflict* occurs between g_1 and g_2 when

$$(name(g_1) = name(g_2)) \wedge (intro(g_1) \neq intro(g_2)).$$

Moreover, this implies that c has some superclasses c', c_1, $c_2 \in X$, such that

$$(c \preceq c' \prec_d c_1) \wedge (c' \prec_d c_2) \wedge (g_1 \in G_{c_1} \backslash G_{c_2}) \wedge (g_2 \in G_{c_2} \backslash G_{c_1}).$$

If $c \prec c'$, then the conflict in c results from an original conflict in c' that is "inherited" by c.

Hence, there is a global property conflict when a class knows two distinct global properties with the same name, and Constraint 2.3 implies that such a conflict is always caused by multiple inheritance.

Conflictless hierarchies. When there is no global property conflict, for instance in a legal model in single inheritance, the restriction of the function *name* to $G_{\uparrow c}$ is injective for all $c \in X$. Hence, in the same conditions, Constraint 2.3 implies that the restriction of *name* to G_c is also injective. Therefore, in the class c context, the identifier of a global property is unambiguous. Of course, *name* is not constrained to be injective throughout its domain G, which would require the CWA. Hence, it must be disambiguated by the context, i.e. the static type of the receiver. Global property conflicts will be examined in Section 3.1.

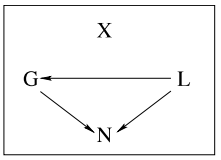
2.4.2. Local properties

Given a class c, L_c denotes the set of local properties *defined* in c and, conversely, the function *def* associates with each local property the class in which it is defined:

$$L = \biguplus_{c \in X} L_c \qquad \text{with} \qquad L_c \triangleq def^{-1}(c). \tag{8}$$

The correspondence between local and global properties is based on their names. Hence, the metamodel must be constrained to enforce this correspondence and make the triangular diagrams commute.

Constraint 2.5 (*Name Triangle*)**.** *The glob function associates a global property with each local property, such that both have the same name:*



$$\forall l \in L,\ name(glob(l)) = name(l).$$

Moreover, it does not make sense to define more than one local property for some global property in the same class; hence we have the following constraint.

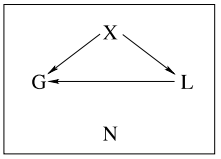
Constraint 2.6 (*Single Local Definition*)**.** *For all $c \in X$, the restriction of glob to L_c is* injective*. Equivalently, for all $g \in G$, the restriction of def to $glob^{-1}(g)$ is injective.*

Therefore, if there is no global property conflict, the restriction of *name* to L_c is also injective. Thus, determining the global property associated with a local one is unambiguous when the name of the global property is unambiguous:

$$\forall l \in L_c, \forall g \in G_c,\ name(l) = name(g) \Longrightarrow glob(l) = g. \tag{9}$$

Of course, *glob* and *name* are not injective throughout their whole L domain. Hence local property names must be disambiguated by the context, i.e. the enclosing class, at compile time, and local properties must be selected by the late-binding mechanism at run time (see below).

Constraint 2.7 (*Class Triangle*)**.** *The global property associated with a local one must be known by the defining class, and all global properties must have been introduced by a local property definition:*



$$\forall c \in X,\ G_{+c} \subseteq glob(L_c) \subseteq G_c.$$

This last constraint closes the upper triangle and achieves the definition of a *legal model*. If a property is considered *abstract* (also known as *deferred*) in its introduction class (i.e. if it has no default implementation), then an abstract local property must still be provided.

Definition 2.8 (*Legal Model*)**.** A *legal model* of a *class hierarchy* is a model which satisfies all Constraints 2.3–2.7.

It follows from Constraint 2.6 that, in a legal model, the function $lid : L \to G \times X$ that maps a local property $l \in L$ to the pair

$$lid(l) \triangleq (glob(l), def(l)) \tag{10}$$

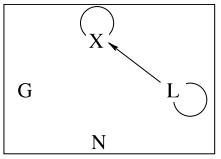
is injective. The syntax `A::foo::B` used in Section 2.3.2 represents *lid*.

Finally, one can formally define *property redefinition*. A local property belongs to an inherited or newly introduced global property. Let $L_{\uparrow c}$ and L_{+c} be the corresponding sets.

$$L_c = L_{\uparrow c} \uplus L_{+c} \quad \text{with} \begin{cases} L_{\uparrow c} & \triangleq L_c \cap glob^{-1}(G_{\uparrow c}), \\ L_{+c} & \triangleq L_c \cap glob^{-1}(G_{+c}). \end{cases} \tag{11}$$

Moreover, *glob* is a one-to-one correspondence between L_{+c} and G_{+c}.

Definition 2.9 (*Property Redefinition*)**.** *Property redefinition* (also known as *overriding*) is defined as the relationship $\ll^{\mathcal{H}}$ ($\ll$ for short) between a local property in $L_{\uparrow c}$ and the corresponding local properties in the superclasses of c:



$$\forall l, l' \in L,\ l \ll l' \overset{\Delta}{\Longleftrightarrow} glob(l) = glob(l') \wedge def(l) \prec def(l').$$

This is a strict partial order, and $\ll_d$ denotes its transitive reduction.

2.4.3. Class definition and model construction

A model of a hierarchy is built by successive class definitions under the OWA. Starting from an empty model, each class definition is first checked for correctness, i.e. the updated model must be legal. The model is then updated according to the following protocol.

Definition 2.10 (*Class Definition*)**.** A *class definition* is a triplet ⟨`classname`, `supernames`, `localdef`⟩, where `classname` is the name of the newly defined class, `supernames` is the set of names of its direct superclasses, which are presumed to be already defined in X, and `localdef` is a set of local property definitions.

A local property definition involves a property name (in a general sense, i.e. a signature if static overloading is considered) and other data, e.g. code, which are not needed here.

Let $\mathcal{H}$ be a legal class hierarchy; then a class definition in $\mathcal{H}$ will produce another hierarchy $\mathcal{H}'$ that extends $\mathcal{H}$. The operational semantics of the metamodel is given by the *meta-object protocol* which determines how this class definition is processed. We informally sketch this four-step protocol as follows. Each occurrence of "new" denotes the instantiation of a class in the metamodel.

1. *Hierarchy update.* A *new* class c with name `classname` is added to X, i.e. $X' = X \uplus \{c\}$. Of course, `classname` must not be the name of a class in X. For each name $n \in$ `supernames`, let d be the corresponding class; then a pair (c, d) is added to $\prec_d$. The correctness of `supernames` must first be checked: (i) it is a set, as it does not make sense to inherit more than once from a given class; (ii) the corresponding classes must already be defined; and (iii) transitivity edges are forbidden. Constraints (i) and (iii) are, however, contrary to the specifications of many languages (see Sections 4.2 and 6.1.2).
2. *Global property inheritance.* $G'_{\uparrow c}$ is computed (4) and global property conflicts are checked (Definition 2.4).
3. *Local definitions.* For each definition in `localdef`, a *new* local property is created, with its corresponding name—this yields L'_c. $L'_{\uparrow c}$ is determined by (11). Then, G'_{+c} is constituted as the set of *new* global properties corresponding to each local property in $L'_{+c} = L'_c \backslash L'_{\uparrow c}$. L'_c and G'_{+c} are then respectively added to L and G—i.e. $L' = L \uplus L'_c$ and $G' = G \uplus G'_{+c}$. Here, again, the names of all local properties are checked for correctness (Constraint 2.6). Ambiguities resulting from global property conflicts are discussed in Section 3.1.
4. *Local property inheritance.* Finally, the protocol proceeds to local property inheritance and checks conflicts for all inherited and not redefined properties, i.e. $G'_{\uparrow c} \backslash glob(L'_{\uparrow c})$. This will be detailed in Section 2.5.

The metamodel is complete, in the sense that all components in Definition 2.2, together with Constraints 2.3–2.7 and Eqs. (3)–(11), are sufficient to characterize a legal model as long as there is no global property conflict. All such legal models could be generated by Definition 2.10. Indeed, given a legal class hierarchy $\mathcal{H}$, for all c in X ordered by some linear extension of $\preceq$, $\langle name(c), \{name(c') \mid c \prec_d c'\}, name(L_c)\rangle$ forms a legal class definition in the context of the hierarchy resulting from previous definitions.

2.5. Local property inheritance and method invocation

So far, we have presented a static model that considers classes and properties at compile time. These classes behave as usual at run time, i.e. they have instances which receive and send messages according to the Smalltalk metaphor. Run-time objects and their construction are not explicit in the model; for instance, we have not merged the proposed metamodel within the ObjVLisp kernel. This is actually not required, since message sending can be modeled at the class level. Indeed, it depends only on the *receiver dynamic type*, which is some class. The specific receiver does not matter, and all direct instances of a given class are equivalent. This explains why method invocation can be compiled and efficiently implemented.

Local property inheritance is a matter of selection of a single local property in a given global property, according to the dynamic type of the receiver. Hereafter, we only detail the method case, which concerns all languages, but it would also apply, in a simplified way, to attributes in languages like Clos and Eiffel which provide attribute redefinition. Method invocation usually involves two distinct mechanisms, namely late binding (also known as message sending or dynamic binding) and calls to `super`. Late binding involves selection of a local property inherited by the considered class, i.e. the dynamic type of the receiver. This constitutes the second level of inheritance, namely *local property inheritance*.

Definition 2.11 (*Local Property Inheritance*)**.** Let $c \in X$ be a class and $g \in G_c$ be a global property. Then the functions $loc, spec : G \times X \to 2^L$ are defined by

$$loc(g, c) \triangleq \{l \in glob^{-1}(g) \mid c \preceq def(l)\} \tag{12}$$

$$spec(g, c) \triangleq \min_{\ll}(loc(g, c)). \tag{13}$$

Hence, $loc(g, c)$ denotes the local properties potentially inherited by c for g, and $spec(g, c)$ denotes the most specific ones with respect to the redefinition relation $\ll$. Late binding involves selecting a local property of g in these sets. The function $sel : G \times X \to L$ denotes the selected property:

$$sel(g, c) = \begin{cases} l, & \text{if } \ spec(g, c) = \{l\} \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{14}$$

According to the definition, *sel* returns the *single most specific* property, i.e. the single element in $spec(g, c)$, but it is unique only in single inheritance. In multiple inheritance, conflicts may occur.

Definition 2.12 (*Local Property Conflict*)**.** Given a class $c \in X$ and a global property $g \in G_{\uparrow c}$, a *local property conflict* occurs when $|spec(g, c)| > 1$. The *class conflict set* is defined as the set

$$cs(g, c) \triangleq def(spec(g, c)) = \min_{\preceq}(def(loc(g, c))) \tag{15}$$

which contains all superclasses of c that define a local property for g and are minimal according to $\preceq$.

We shall examine local property conflicts and the reasons for this definition in Section 3.2. All functions *loc*, *spec*, *sel*, and *cs* are partial, and are defined only on $g \in G$ and $c \in X$ when $g \in G_c$.

Method invocation can also use a secondary mechanism called "call to `super`". It allows a local property l to call another one, say l', which is redefined by l, i.e. $l \ll l'$. We keep the term "`super`" used in Smalltalk and Java as it is the most popular, but we specify it with a slightly different meaning. Here it is closer to Eiffel `Precursor`. The point is that l and l' belong to

the same global property, i.e. $glob(l) = glob(l')$, while Smalltalk and Java accept `super.foo` in method `bar`. The syntactic status of `super` is also questionable. In Smalltalk and Java, its variable-like syntax, similar to `self` (or `this`), seems to denote a fictitious object, namely `self` "in the superclass". In Eiffel, `Precursor` and `Current` also have the same syntax as symbols with a single capital letter. In contrast, like Clos, we consider that `super` and `call-next-method` are functions, i.e. the current global property in the superclass.

Definition 2.13 (*Call to `super`*)**.** Let $l \in L$ be a local property. The function $supl : L \to 2^L$ is defined by

$$supl(l) \triangleq \{l'' \mid l \ll l''\}. \tag{16}$$

Then call to `super` involves calling the local property l' that is the single member of $\min_{\ll}(supl(l))$.

Like late binding, call to `super` involves selection of the most specific property, i.e. the single local property l' which satisfies $l \ll_d l'$. However, l' is only unique in single inheritance, where l' is determined by the uniqueness of c' such that $def(l) \prec_d c'$; then $l' = sel(glob(l), c')$.

The bottom-up call to `super` is not the only way of combining methods. A top-down mechanism has also been proposed in Lisp-based object-oriented languages (Flavors, Clos) under the name `:around` methods (also known as *wrappers*) and in Simula and Beta under the name `inner`. Goldberg et al. [48] propose their integration in a single language. These mechanisms present similar problems in case of multiple inheritance. We shall examine them in Section 3.2 together with local property conflicts.

3. Multiple inheritance conflicts

Conflicts are the main difficulty in multiple inheritance. They are usually expressed in terms of name ambiguities. The metamodel yields a straightforward analysis in terms of reified entities and distinguishes between two kinds of conflict which require totally different answers. The following analysis is mostly the same as that in [36], just enhanced with the metamodel.

3.1. Global property conflict

According to Definition 2.4, a *global property conflict* (called "name conflict" in [36]) occurs when a class specializes two classes having distinct but homonymic global properties. Fig. 3 shows two global properties named `department`. The first one specifies a department in a research laboratory. The other specifies a teaching department in a university. It is then expected that the common subclass `Teacher-Researcher` inherits all the different global properties of its superclasses. However, the name `department` is now ambiguous in two situations: (i) when redefining `department` in `Teacher-Researcher`; and (ii) when sending the message `department` to a receiver statically typed by `Teacher-Researcher`. Anyway, this situation is simply a naming problem. It must be solved and a systematic renaming in the whole program would solve it. Different answers are possible, which do not depend on the specific kind of properties, i.e. attributes, methods, or types, since the conflict only involves names.

Nothing, i.e. error. The language does not specify any answer to such a conflict but it signals an ambiguity. This forces the programmer to rename at least one of the two properties, but this must be done throughout the program, and hence under the CWA. This can be error prone or even impossible, when the source code is unavailable.

Fully qualified names. This simply involves an alternative unambiguous fully qualified syntax, which juxtaposes the property name with the name of a class in which the property name is not ambiguous, for instance the class that introduces the global property. In the example, `Teacher:department` would denote the global property known as `department` in the `Teacher` class. This naming would be unambiguous since, in a legal model (Definition 2.8), *gid* is injective (7). A similar solution is possible for attributes in C++ with the *scope resolution operator* ::.[3] Note that, with fully qualified names, a global property conflict requires a solution only when the programmer wants to use the ambiguous name, in a context where it is ambiguous. Hence, this is a modular (OWA) and lazy solution.

At a call site, fully qualified names can be replaced by explicit *upcasts* (provided that the considered language offers such a syntactic sugar), in a way similar to static overloading disambiguation (page 559). Indeed, `x.Teacher:department` is equivalent to `(Teacher)x.department` (in a Java-like syntax) when the static type of `x` is `Teacher-Researcher`. However, the upcast syntax cannot be used for defining, i.e. redefining, `Teacher:department`. C# provides a solution based on a qualification for method definitions and upcast for call sites (Section 6.1.2).

Local renaming. Local renaming changes the designation of a property, both global and local, in a class and its future subclasses. In the `Teacher-Researcher` class of the example, `department` inherited from `Teacher` can be renamed as `teach-dept`, and `department` inherited from `Researcher` as `res-dept`. Thus, `department`

[3] In C++, attributes cannot be redefined—this would be static overloading, and hence a new global property. Therefore, a single local attribute corresponds to each global attribute and :: denotes both the local or the global one. For methods, :: corresponds to a static call and denotes a local property.

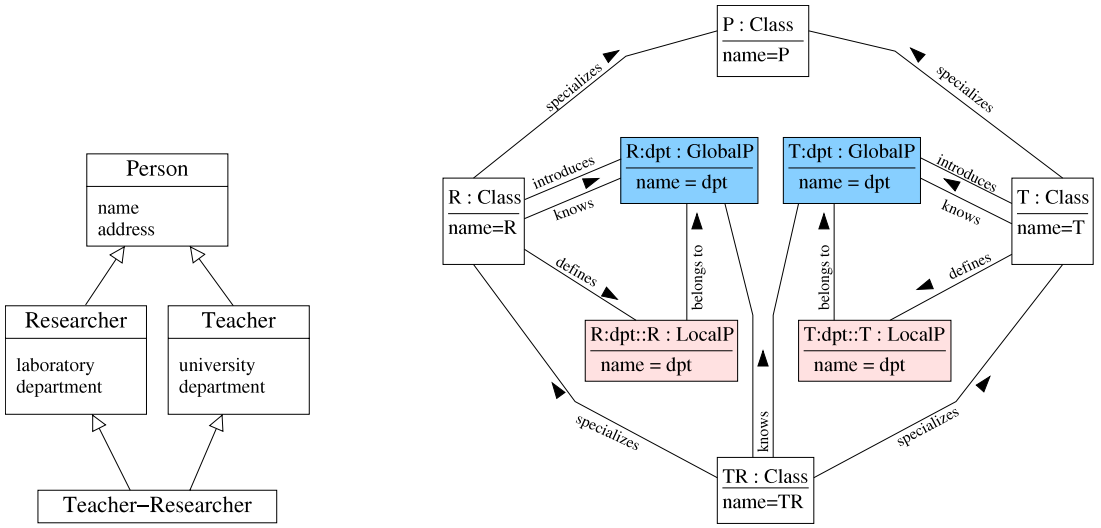


Fig. 3. Global property conflict. The class diagram (a) depicts a conflict between the two properties named `department` introduced in two unrelated classes and the instance diagram (b) shows the corresponding metamodel instantiation. All entities are tagged by unambiguous fully qualified names, and the names are abbreviated.

denotes, in `Researcher`, the same global property as `res-dept` in `Teacher-Researcher`. Conversely, in the `Teacher-Researcher` class, `res-dept` and `teach-dept` denote two distinct global properties. Eiffel provides the `rename` keyword for this. Renaming is also modular (OWA) but no longer lazy, since it is required even when the programmer does not use the name in an ambiguous context (i.e. when redefining one of the conflicting properties or calling it on a receiver typed by the class considered). Moreover, as class hierarchies are not forced to form lattices (i.e. two unrelated classes do not have a single lower bound), the same conflict can occur in different subclasses, with possibly different renamings. To further confuse the issue, this might lead to subsequent *renaming conflicts*, when the same global property is inherited in a class with different names. This situation is, however, not worrying, and Eiffel just requires the programmer to rename the property until it is known by a single name in the class considered.

Unification, i.e. silence. Dynamic languages like Clos consider that if two global properties have the same name then they are not distinct. Java has the same behavior when a class implements two interfaces which introduce a global property with the same signature (Java packages provide indeed a fully qualified syntax for classes, not for properties). Hence, the global property conflict is not recognized, and multiple inheritance ambiguities are deferred to local property inheritance. This solution is thus very close to the first one, and it does not allow the programmers to express their intention with respect to distinct global properties unless there is global renaming. In Fig. 3, the two departments represent distinct concepts. If the programmers' intention is a single concept, then they should have defined a common superclass introducing a single global property for this concept. However, silence adds an extra flaw, as the programmers may be unaware of the problem or might misunderstand it.

The C++ case. In contrast with the aforementioned languages which provide a uniform behavior for global property conflicts, C++ has a dual behavior for methods. If the conflicting method `department` is redefined in the conflicting class `Teacher-Researcher`, then both properties are unified. Without redefinition, each definition remains reachable through an upcast. Therefore a limited, manual form of local renaming is possible, by introducing two global properties, with local definitions calling respectively the original ones, through appropriate upcasts (Fig. 4). However, these new methods do not behave as true redefinitions, and this manual renaming is not equivalent to Eiffel's one.

Naming convention. When languages deal with global property conflicts by signalling an error or unifying conflicting properties, the situation may have no solution. Global renaming is the only way out, but this might be impossible if the source code of the conflicting classes is not available, or if the conflicting classes are contractually used by other people. Therefore, it is necessary to anticipate conflicts by enacting and following a naming convention, for instance by prefixing property names with all or part of the introducing class name. The naming convention differs from full qualification by its subjectiveness as it relies only on the programmer's discipline instead of being enforced by the language specifications. It also differs by its uniformity, since the convention must be used everywhere, while a qualification is required only in a conflict context. Anyway, in dynamic typing, this is the only solution since the metamodel cannot distinguish between homonymic properties. In static typing, this convention would be redundant with static types and would make the code clumsy.

```
class B: public virtual A {
    public:
    virtual void foo () {...}}

class C: public virtual A {
    public:
    virtual void foo () {...}}

class D: public virtual B, public virtual C {
    public:
    virtual void foob () {
        static_cast<B*>(this)->foo()}
    virtual void fooc () {
        static_cast<C*>(this)->foo()}}

int main(int argc, char* argv[]){
   D* d = new D();
   //d->foo();  error: request member 'foo' is ambiguous
   d->foob(); //calls B::foo
   d->fooc(); //calls C::foo
   B* b = d;
   b->foo();  //does not call foob}
```

In C++ (left), defining `foo` in `D` would unify the two global properties introduced in `B` and `C`. In contrast, the `foob`/`fooc` definitions resemble local renaming. The original definitions can be called through these new names on all subclasses of *D*. However, this renaming is asymmetric, and hence illusory. Indeed, calling `foo` on an instance of `D` typed by `B*` does not call `foob`. Therefore, contrary to Eiffel renaming (right), `foob` does not behave as a redefinition of `foo`. This example has been tested with `g++ 4.4.3`, with and without `-ansi -pedantic` options.

```
class B
    inherit A end
    feature foo is do ... end
end

class C
    inherit A end
    feature foo is do ... end
end

class D
    inherit
        B rename foo as foob redefine foob end
        C rename foo as fooc end
    feature
        foob is do ... Precursor ... end
end

class TEST
  feature main is
    local   d: D   b: B
    do
      create d
    --d.foo error: unknown feature 'foo' in type D
      d.foob  --calls foo in B through Precursor
      d.fooc  --calls foo in C
      b := d
      b.foo   --calls foob in D
    end
end
```

Fig. 4. Local renaming in C++ (left) and Eiffel (right). `A`, `B`, `C`, `D`, and `foo` are shortcuts for the example in Fig. 3.

As a provisional conclusion, global property conflicts represent a minor problem. They could easily be solved in any programming language at the cost of a cosmetic modification (namely, *qualifying* or *renaming*), and they should not be an obstacle to the use of multiple inheritance. Both solutions require a slight adaptation of the metamodel. This is straightforward in the case of full qualification. All names in the class definition, more generally in the program text, can be simple or fully qualified. An analogous syntax is available in all languages where name-spaces are explicit, e.g. packages in Java or Common Lisp, but it does not apply at the right level. The adaptation is less simple for local renaming, since the function $name_{\mathcal{H}}$ is no longer global and must now take two parameters, i.e. the property and class. Renaming clauses must also be integrated in Definition 2.10. The best solution might be to combine both qualification and renaming, with the latter being restricted to a lexical scope. Full qualification is the default solution, but it might be clumsy if the fully qualified name is overused in a compilation unit. Therefore, the programmer might prefer to rename it, but the renaming scope is the compilation unit, and renaming is no longer inherited.

3.2. Local property conflict

3.2.1. Conflict definition and masking

According to Definition 2.12, a *local property conflict* (called "value conflict" in [36]) occurs when a class inherits two local properties from the same global property, with none of them more specific than the other with respect to the redefinition relation $\ll$ (Definition 2.9). Fig. 5a illustrates this situation with two classes, `Rectangle` and `Rhombus`, with both redefining the `area` method which was introduced in a common superclass, `Quadrilateral`. In the common subclass `Square`, no properties are more specific than the other. Which one must be selected? Fig. 5c depicts the model of the example, restricted to the concerned property. To be compatible with the syntax for global properties, here we adopted a syntax similar to that of C++ but inverted, namely `area::Rhombus` (instead of `Rhombus::area`) denotes the local property `area` defined in the `Rhombus` class. It is always possible in a legal model since the *lid* function (10) is injective. This is the same meaning as the C++ *scope resolution operator* :: (see Footnote 3, page 566), apart from operand inversion. If the global property name is ambiguous, full qualification can lead to `Quadrilateral:area::Rhombus`. Note, however, that this is metalanguage syntax and is not necessarily part of the considered programming language. In the following example, all class names from Fig. 5 are abbreviated as in Fig. 5c. Consider the `area` global property introduced in `Q`, denoted by `Q:area`, and the class `S`. Then, according to Definitions 2.11 and 2.12,

$$\begin{aligned} loc(\texttt{Q:area}, \texttt{S}) &= \{\texttt{Q:area::Q}, \texttt{Q:area::Rh}, \texttt{Q:area::Re}\},\\ spec(\texttt{Q:area}, \texttt{S}) &= \{\texttt{Q:area::Rh}, \texttt{Q:area::Re}\},\\ cs(\texttt{Q:area}, \texttt{S}) &= \{\texttt{Rh}, \texttt{Re}\}. \end{aligned}$$

The conflict follows from the fact that $spec(\texttt{Q:area}, \texttt{S})$ has two most specific elements. It vanishes when the definition of one of them, say `Q:area::Rh`, is removed, though `Rh` still inherits a local property `area` from `Q` (Fig. 5b). However, some

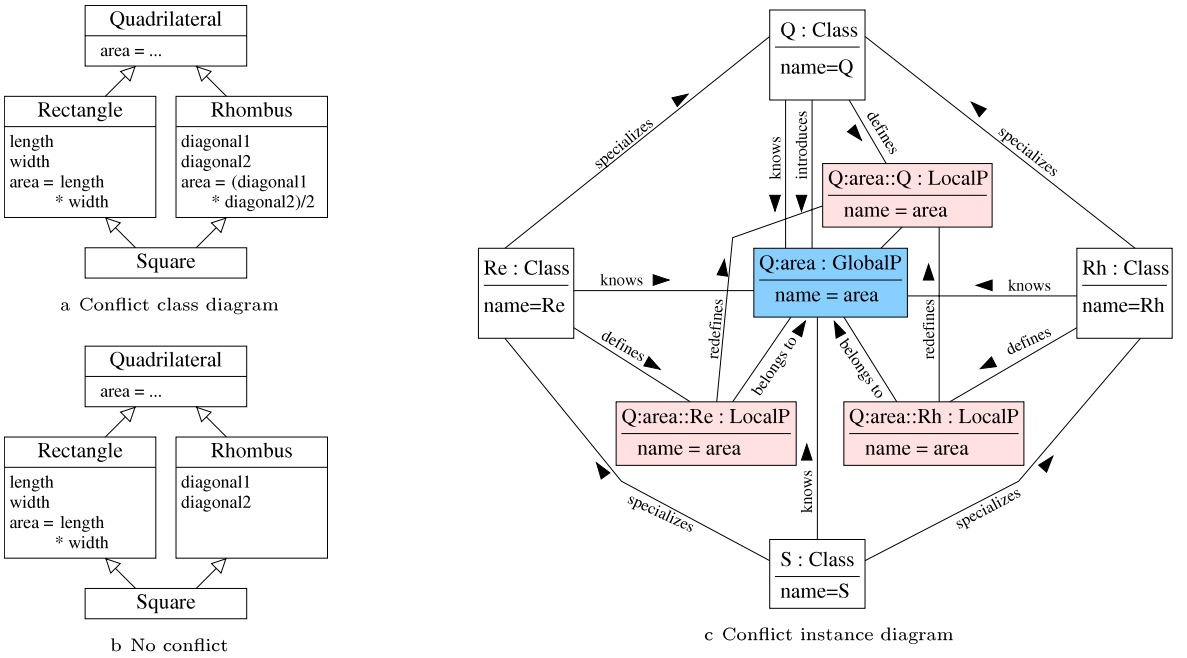


Fig. 5. Local property conflict. The class diagram (a) depicts a conflict between the two local properties `area` redefining the same property in two unrelated classes and the instance diagram (c) shows the corresponding metamodel instantiation. In contrast, there is no conflict in class diagram (b).

languages like Eiffel consider that there is still a conflict in `S`, between the property defined in `Re` and that inherited by `Rh`, unless the latter is abstract. Therefore, it is important to understand why we choose this conflict definition. Apart from redefinition, specialization is inherently *monotonic*. That is, in the definition of A, what is true for an instance of A is also true for an instance of any subclass of A. This goes back to Aristotelian syllogistic—see Section 2.1. In contrast, property redefinition entails *non-monotonicity*, in the sense of *non-monotonic* (also known as *defeasible*) inheritance theories [99,52]. A local property is a *default value* for instances of the class that defines it. For all instances of a subclass A', the redefining property *masks* the redefined one. Redefinition must thus be understood as follows.

Requirement 3.1 (*Masking Rule*)**.** *Let g be a global property, and l a local property of g defined in class A. Then l implements g for all instances of A,* unless otherwise stated, *i.e. unless the considered object is an instance of a subclass of A which redefines g. If $A' \prec A$ and l' redefines l in A' ($l' \ll l$), l' will then* mask *l for all instances of A', direct and indirect alike. This remains true if l' is in turn masked by a third property l'' defined in a subclass A'' of A'.*

Therefore, in spite of the non-monotonicity which is inherent to redefinition, masking is strictly monotonic, and Definition 2.12 states only that a class cannot inherit a masked local property. In the example considered in Fig. 5b, `area`::`Q` implements `area` for all instances of `Q`, except instances of `Re`, and `area`::`Re` implements it for all instances of `Re`, including those of `S`. This means that *defining* a property is stronger than *inheriting* it, since an inherited property does not mask anything from the inheriting class [33]. In this respect, method combination (Section 3.2.3) is a way to recover some monotonicity; if l' calls `super`, all instances of A' will behave like those of A, along with some extra behavior. This fosters *behavioral subtyping* [66].

3.2.2. Conflict solutions

Unlike the global property conflict, there is no intrinsic solution to the local property conflict. Consequently, either the programmer or the language must bring additional semantics to solve the conflict, and this additional semantics may depend on the kind of property, i.e. attribute, method, or type. There are roughly three ways to do this.

Nothing, i.e. error. The language considered does not specify any answer to local property conflicts but it signals an error at compile time. This forces the programmer to define a local property in the class where the conflict appears. In this redefinition, a call to `super` is often sought, but it will be ambiguous (see below). A variant of this approach makes the class which introduces the conflict (`Square`) *abstract*, by implicitly defining an *abstract* local property[4] (`area`) in this class [77]. This forces the programmer to define the property in all direct non-abstract subclasses.

4 With *abstract local properties*, the conflict definition must be slightly adapted. An actual conflict occurs when $spec(g, c)$ contains several non-abstract properties. If all members of $spec(g, c)$ are abstract, the inherited property in c is also abstract.

Selection. The programmer or the language arbitrarily select the local property to inherit. In many dynamic languages, the choice is made with a *linearization* (Section 4); in Eiffel, the programmer can select the desired property with the `undefine` keyword.

Combining. Local properties are objects, and hence are composed of some metaproperties. Solving the conflict thus amounts to combining the conflicting objects and, for each *metaproperty*, selecting or combining. This is, for instance, quite explicit in Clos *slot-definition* specifications. Therefore, for some values or particular properties (especially for some metaproperties), the conflict must be solved by combining the conflicting values. For instance, in Java, when the conflict concerns the declared method exceptions, it should be solved by taking the intersection of all declared exceptions. Another example is the return and parameter types of method. This will be examined in Section 5. Combining is also needed by Eiffel contracts with disjunction of preconditions and conjunction of postconditions. Generally speaking, *method combination* is often the solution when several methods conflict, which is examined hereafter.

Overall, the solution is *redefining*, *selecting*, or *combining*, or a mix, since redefinition can be combined with selection. For instance, in C++, selection must be done by redefining the local property with a static call (done with the :: operator) to the selected one.

3.2.3. Call to `super` and method combination

Calls to `super` present a similar but more general kind of conflict. Let $g \in G$ be the considered global property, $c \in X$ be the receiver dynamic type, and $l \in loc(g, c)$ be the currently invoked local property that calls `super`. Suppose first that l has been determined without local property conflict. This means that $spec(g, c) = \{l\}$ and $loc(g, c)\backslash\{l\} = supl(l)$ (see Definitions 2.11 and 2.13). In other words, all other local properties that might be combined are in $supl(l)$. Therefore, the situation is exactly the same as late binding, except that the selection or combination process must now consider $supl(l)$ instead of $loc(g, c)$. Then the set $\min_{\ll}(supl(l)) = \{l' \mid l \ll_d l'\}$ may not be a singleton, thus making `super` as ambiguous as a local property conflict. This is the case if l is the `area`::`Square` method that has been defined to solve the local property conflict in Fig. 5a. A solution is to consider that `super` is legal only when $\min_{\ll}(supl(l))$ is a singleton. If this is not a singleton, an explicit selection among the conflicting local properties is required, and `super` must be qualified by some discriminating class. A syntactic alternative would be a static call, as with :: in C++. We exclude, however, static calls because they explicitly mention the property name, like `super` in Smalltalk and Java. Hence, the C++-like syntax `area`::`Rhombus` would be replaced by `super`⟨`Rhombus`⟩, but with this meaning only within the code of an `area` method of a `Rhombus` subclass. Eiffel, with `Precursor` instead of `super`, provides similar specifications. Qualified `super` and static calls have, however, a major drawback, as they may yield *multiple evaluations* of the local property defined in the diamond root (Fig. 5). For instance, consider a `foo` method with a local definition in each of the four classes, with each definition statically calling all local properties defined in its direct superclasses.

In a conflict case, when the current local property l has been arbitrarily selected in $spec(g, c)$, e.g. if l is `area`::`Rhombus`, the point is that $supl(l)$ does not include all other local properties in $loc(g, c)$. Now, $spec(g, c)\backslash\{l\}$ and $supl(l)$ are disjoint non-empty sets. Hence, call to `super` cannot combine all inherited local properties. Finally, calls to `super` can also occur in a local property which has been invoked by a call to `super`, not by a "primary" late binding, but this does not lead to further complication.

Overall, there are three possibilities: (i) a constrained, unqualified keyword `super`, only sound when there is no conflict in $supl(l)$; (ii) a qualified use of `super`, which allows the programmer to explicitly reference the class when there is a conflict, e.g. `super`⟨`Rhombus`⟩; and (iii) a third mechanism, called `call-next-method` and based on linearizations, which avoids the aforementioned drawbacks of `super`; it is discussed hereafter. All three mechanisms are compatible with each other. A language can provide all of them. Linearizations are more flexible, whereas unqualified `super` has a restricted use and its qualified version can lead to multiple evaluations.

4. Linearizations

Linearizations were introduced in the early 1980s in Flavors, an object-oriented extension of Lisp [105]. They have been widely used in many dynamically typed object-oriented languages such as Loops and Common Loops [93,11], Clos [92], Yafool [28], Power Classes [57], Dylan [87], Python [101], etc. We know only a single use in a static typing and full multiple inheritance context, namely in C++, where linearizations apply to constructors and destructors [54]. They are, however, also used in statically typed mixin-based languages, such as Gbeta [40] and Scala [78] (see Section 6.2).

4.1. Principle

The linearization principle involves computing, for each class $c \in X$, a total ordering on the set of superclasses of c, i.e.

$$supc(c) \triangleq \{c' \mid c \preceq c'\}. \tag{17}$$

This ordering is called the *class precedence list* in Clos and the *method resolution order* in Python.

Definition 4.1 (*Class Linearization*)**.** Given a class hierarchy $\mathcal{H}$, a *class linearization* is defined as a function $clin^{\mathcal{H}} : X \to (X \to \mathbf{N})$, i.e. *clin* for short, such that $clin(c)$ is a *permutation* of $supc(c)$, i.e. a bijective function $clin(c) : supc(c) \to 1..|supc(c)|$. For the sake of readability, $clin(c)$ is hereafter denoted $clin_c$. An additional constraint is that $clin_c(c) = 1$ for all c.

Hence, a linearization yields a total order $(supc(c), \preceq_{clin(c)})$, whereby $x \preceq_{clin(c)} y \overset{\Delta}{\iff} clin_c(x) \leq clin_c(y)$. An alternative notation is as follows: $clin(c) = (c_1, \ldots, c_k)$, with $c = c_1$, $supc(c) = \{c_i \mid i \in 1..k\}$ and $clin_c(c_i) = i$ for all $i \in 1..k$.

Class linearizations involve only the poset $(X, \preceq)$ and are just dedicated to solving local property conflicts. Therefore, linearizations must be mapped from classes to local properties, i.e. from the poset $(supc(c), \preceq)$ to the poset $(loc(g, c), \ll)$, for each global property $g \in G_c$.

Definition 4.2 (*Local Property Linearization*)**.** Given a class hierarchy $\mathcal{H}$ equipped with a class linearization *clin*, a *local property linearization* is defined by the function $llin : G \times X \to (L \to \mathbf{N})$ such that $llin(g, c) = (l_1, \ldots, l_m)$, with $loc(g, c) = \{l_i \mid i \in 1..m\}$ and $1 \leq i < j \leq m \Rightarrow clin_c(def(l_i)) < clin_c(def(l_j))$. It yields a total order $(loc(g, c), \preceq_{llin(g,c)})$ analogous to $(supc(c), \preceq_{clin(c)})$.

Note that *def* is monotonic from $(L, \ll)$ to $(X, \preceq)$ and $def(loc(g, c)) \subset supc(c)$. Hence, *def* is also monotonic from $(loc(g, c), \preceq_{llin(g,c)})$ to $(supc(c), \preceq_{clin(c)})$.

Definition 2.11 must be modified accordingly. Now, the selection function *sel* selects the first property in this ordering, i.e. $sel(g, c) = l_1$, and call to `super` is carried out by the `call-next-method` mechanism.

Definition 4.3 (*Call Next Method*)**.** The *call-next-method* mechanism relies on the partial function $cnm : X \times L \to L$ such that, with the previous definition notations, $cnm(c, l_i) = l_{i+1}$ when $i < m$, and $cnm(c, l_m)$ is undefined.

Instead of calling the local property in the superclass like `super`, it calls the next method in the receiver's linearization. When used for combination, linearizations can thus combine all inherited methods while avoiding multiple evaluations which may occur with qualified `super`. However, it is essential to note that, in the expression $cnm(c, l_i)$, c is not the class which defines l_i, i.e. $def(l_i)$, but rather the receiver's dynamic type; hence $c \preceq def(l_i)$. Single inheritance ensures that $l \ll cnm(c, l)$ for all $c \preceq def(l)$, but this is no longer verified once there is a local property conflict. Therefore, the existence of this next method cannot always be statically (i.e. when compiling $def(l_i)$) ensured, and an auxiliary function `next-method-p` is needed to allow the programmer to check it at run time. However, when the linearization is a linear extension (see hereafter), this run-time check is only required when the method has been declared *abstract* in $def(l_i)$ superclasses.

Finally, `call-next-method` could also be considered top-down. In Simula and Beta, `inner` is restricted to single inheritance, but Clos wrappers, i.e. `:around` methods, are integrated with `call-next-method`. Actually, the same `call-next-method` function works top-down in wrappers and bottom-up in ordinary methods. In contrast, constrained or qualified `super` cannot work top-down, since there is no static way to deal with conflicting subclasses. Thus, we shall not develop a top-down version for `call-next-method`, since it is exactly like the bottom-up mechanism, but in reverse order.

4.2. Requirements

Several theoretical studies have determined what should be a "good" linearization. We review here their main conclusions. All proofs can be found in the papers referenced.

Linear extensions. In order to ensure that the selection respects the *masking rule* (Requirement 3.1), i.e. that no other property is more specific than the selected one, the total order $(supc(c), \preceq_{clin(c)})$ must be a *linear extension* (also known as *topological sorting* [62]) of $(supc(c), \preceq)$ [32]. This means that

$$c \preceq c' \preceq c'' \Longrightarrow clin_c(c') \leq clin_c(c'') \tag{18}$$

or, equivalently, that the restriction $\preceq / supc(c)$ is a subset of $\leq_{clin(c)}$ for all $c \in X$. This implies that the selected property is taken from the *class conflict set*, i.e. $def(sel(g, c)) \in cs(g, c)$ [36]. This requirement is easy to meet and is fulfilled in most recent languages (it was actually satisfied in mid-1980s languages [93,11,74,92]), with the notable exception of Python, which uses a simple depth-first search (for "classic classes" only). When there is no local property conflict and linearizations are used for selection, linear extensions have the desired behavior according to Definition 2.11, i.e. they select the single most specific local property. Hence, such linearizations represent only a default selection mechanism, and the programmer can simply switch it off when there is a conflict by redefining the property and explicitly solving the conflict. Hereafter, and unless otherwise stated, we shall consider that all linearizations are linear extensions.

Monotonicity, i.e. linearization inheritance. Another important requirement is that the class linearization should be *monotonic*—i.e. the total ordering of a class extends that of its superclasses [34,35,9,40]. This amounts to inheriting linearizations, i.e. $\leq_{clin(c')}$ is a subset of $\leq_{clin(c)}$, for all $c \prec c'$ in X. Equivalently, this means that

$$c \preceq c' \preceq c'', c''' \Longrightarrow \big(clin_c(c'') \leq clin_c(c''') \iff clin_{c'}(c'') \leq clin_{c'}(c''')\big). \tag{19}$$

Since $clin_c(c) = 1$ (Definition 4.1), a monotonic linearization is always a linear extension. Conversely, when the linearization is a linear extension, (19) is verified for all $\preceq$-related pairs (c'', c'''), and therefore monotonicity constrains only $\preceq$-unrelated pairs.

When the linearization is used for combination, monotonicity makes the order of method invocations preserved by inheritance, and of course *llin* is also monotonic. Furthermore, monotonicity implies a nice property when the linearization is used for selection, namely a class always behaves like at least one of its direct superclasses or, equivalently, *inheritance cannot skip a generation*.

However, the need for monotonicity is not as easy to meet as that for linear extension. Actually, given a class hierarchy $(X, \preceq)$ equipped with a monotonic linearization *clin*, and two classes $c, c' \in X$, it may be impossible to extend the hierarchy to a common subclass of c and c' without losing monotonicity, because $clin_c$ and $clin_{c'}$ conflict on some pair (x, y), i.e. $clin_c(x) < clin_c(y)$ and $clin_{c'}(x) > clin_{c'}(y)$. This *linearization conflict* involves a cycle in the union of $\leq_{clin(c)}$ and $\leq_{clin(c')}$.

Local and extended precedence order. The actual linearization principle is to totally order $\preceq$-unrelated superclasses, especially direct superclasses. As such orderings are rather arbitrary, they are usually explicitly given by the programmer, like the order of superclass declaration. Hence, in Definition 2.10, `supernames` is a *totally ordered* set, called *local precedence order* in Clos. Linearization is thus required to respect these local orders. This is, however, not always possible for the same reasons as for monotonicity. In the following, $lpo(c) = (c_1, \ldots, c_k)$ denotes the local precedence order of class c, i.e. the ordered list of its direct superclasses. Formally, *lpo* should be added to the hierarchy tuple of Definition 2.2.

An *extended precedence order* has also been considered [55]. It is defined as a kind of bottom-up propagation of local precedence orders. It is the third constraint underlying the C3 linearization (see below).

Poset-based, modular, and transitivity-free. We consider that linearizations are based on posets, like inheritance. This has several consequences. The linearization should depend only upon the topology of the inheritance graph. For instance, it should not consider class names, or programmer ages. Moreover, two isomorphic inheritance graphs should have isomorphic linearizations. Linearizations that satisfy this isomorphism requirement are called *language independent* [45]. The linearization must therefore be *context-free*, and hence *modular*. Indeed, language-independent linearizations of an inheritance graph cannot depend on the whole including hierarchy.

Introducing isomorphism of inheritance graph implies formally defining inheritance graphs.

Definition 4.4 (*Inheritance Graph*)**.** Let $\mathcal{H}$ be a class hierarchy and $c \in X^{\mathcal{H}}$ be a class. Then the *inheritance graph of c in* $\mathcal{H}$, is the 4-tuple $\langle c, supc(c), \preceq, lpo\rangle$ uniquely determined by c and $\mathcal{H}$, where the latter two components are restricted to the second one. Let $\mathcal{H}$ and $\mathcal{H}'$ be two class hierarchies, $\langle c, supc(c), \preceq, lpo\rangle$ and $\langle c', supc'(c'), \preceq', lpo'\rangle$ be two inheritance graphs from the respective hierarchies.

Then $f : supc(c) \to supc'(c')$ is an *isomorphism of inheritance graphs* iff

$$f(supc(c)) = supc'(c')$$
$$\forall x, y \in supc(c), x \preceq y \iff f(x) \preceq' f(y) \quad \textit{(hence, } f \textit{ is a poset isomorphism and } f(c) = c')$$
$$\forall b \in supc(c), lpo(b) = (c_1, c_2, \ldots, c_k) \iff lpo'(f(b)) = (f(c_1), f(c_2), \ldots, f(c_k)).$$

Finally, the aforementioned limits of linearizations can be stated in the following concise way.

Proposition 4.5 (*[35,45]*)**.** *There are no language-independent linearizations that are monotonic for all inheritance hierarchies.*

Therefore, in practice, monotonicity might remain a sought but not required property.

There is one last point of contention. The metamodel is only based on sets and posets, and we think that linearizations should not be sensitive to transitivity edges. The programmer might want to declare that $lpo(C) = (A, B)$ while $A \prec B$ (of course, when $B \prec A$, it would contradict the *lpo*). This (C, B) transitivity edge is thus intended to add extra ordering, but it is redundant with $\prec$. Therefore, sensitiveness to transitivity edges is carried out by the underlying algorithms, in a rather counterintuitive way. Nevertheless, most languages, e.g. Clos, Dylan, Python, have adopted transitivity-dependent linearizations, i.e. they do not constrain direct superclasses to be $\prec$-unrelated.

4.3. Some linearizations

Many linearizations have been studied, and a few of them are used in production languages. Hereafter, the presented algorithms assume that there are no transitivity edges.

Notations. A family of linearizations uses two Lisp-like operators: the `cons` operator **.** such that $x \,.\, (y_1, \ldots, y_k) \triangleq (x, y_1, \ldots, y_k)$; and various concatenation (`append`) or merge operators $+$ such that $() + x = x + () = x$. The $+$ operators have higher priority than **.**, and all list operands are without duplicates, since they represent totally ordered sets. In the following, let A be a class and B_i, $i = 1..n$, be its direct superclasses, which are presumed to be $\prec$-unrelated, with $lpo(A) = (B_1, \ldots, B_n)$.

4.3.1. In common loops and C++

This is the simplest linearization which gives a linear extension, but [32,54] present counterexamples showing that it does not meet the other requirements. It was introduced in Common Loops and is used in C++ (for constructors and destructors) and Scala. It can be defined in the following way:

$$clin(A) = A \,.\, clin(B_1) \oplus \cdots \oplus clin(B_n)$$

where $\oplus$ is special concatenation which removes duplicates

$$(a_1, \ldots, a_k) \oplus (b_1, \ldots, b_m) = \begin{cases} (a_2, \ldots, a_k) \oplus (b_1, \ldots, b_m) & \text{if } a_1 \in \{b_1, \ldots, b_m\} \\ a_1 \,.\, (a_2, \ldots, a_k) \oplus (b_1, \ldots, b_m) & \text{otherwise.} \end{cases}$$

The definition is sound because $\oplus$ is associative.

4.3.2. C3 linearization

It has been designed for Dylan but finally not applied to it for compatibility reasons [9]. In Python, it is used for "new-style" classes only. C3 is the only linearization which fulfils all requirements—actually it takes its name from three criteria, namely linear extension, monotonicity, and extended precedence order. Ernst [40] gives the following definition.

$$clin(A) = A \,.\, clin(B_1) \boxplus \cdots \boxplus clin(B_n)$$

where $\boxplus$ is the usual `merge` operator extended to extensional orders as follows:

$$(a_1, \ldots, a_k) \boxplus (b_1, \ldots, b_m) = \begin{cases} a_1 \,.\, (a_2, \ldots, a_k) \boxplus (b_2, \ldots, b_m) & \text{if } a_1 = b_1 \\ a_1 \,.\, (a_2, \ldots, a_k) \boxplus (b_1, \ldots, b_m) & \text{if } a_1 \notin \{b_1, \ldots, b_m\} \\ b_1 \,.\, (a_2, \ldots, a_k) \boxplus (b_2, \ldots, b_m) & \text{if } a_1 \in \{b_1, \ldots, b_m\} \wedge b_1 \notin \{a_1, \ldots, a_k\} \\ \text{impossible otherwise,} & \text{i.e. if } a_1 \in \{b_2, \ldots, b_m\} \wedge b_1 \in \{a_2, \ldots, a_k\}. \end{cases}$$

However, this definition is only sound when $n = 2$ because $\boxplus$ is not associative. The correct definition, i.e. the original definition by Barrett et al. [9], is obtained with an n-ary operator $\boxplus_i$:

$$clin(A) = A \,.\, \boxplus_i clin(B_i).$$

Let $L_i = (a_1^i, \ldots, a_{k_i}^i)$ be p lists to merge, for $i \in 1..p$. Let j be the least integer such that $a_1^j \notin (a_2^i, \ldots, a_{k_i}^i)$ for all $i \neq j$. Then,

$$\boxplus_i L_i = \begin{cases} a_1^j \,.\, \boxplus_i L_i' \text{ if } j \text{ exists,} & \text{where } L_i' = \begin{cases} L_i \text{ if } a_1^j \neq a_1^i \\ (a_2^i, \ldots, a_{k_i}^i) \text{ if } a_1^j = a_1^i \end{cases} \\ \text{impossible otherwise.} \end{cases} \tag{20}$$

If there is no such j, then there is a cycle $n_0, n_1, \ldots, n_q = n_0$ with $1 < q \leq k$, such that $a_1^{n_{i-1}} \in \{a_2^{n_i}, \ldots, a_{k_{n_i}}^{n_i}\}$ for all $0 < i \leq q$. Conversely, if there is a cycle, the algorithm will reach a state where there is no such j.

4.3.3. Other linearizations

Some other linearizations cannot be expressed in terms of concatenation or merging. Clos linearization involves taking successive minimals according to the union of both specialization ($\prec$) and local precedence orders (*lpo*). When there are several minimals, the algorithm selects the single minimal which is a direct superclass of the most recently taken class in the linearization. It is not always monotonic. In [35], the proposed monotonic linearization computes a linear extension of the union of linearizations of all direct superclasses $\cup_i clin(B_i)$, which represents a short cut for $(\cup_i supc(B_i), \cup_i \preceq_{clin(B_i)})$. It then applies the Common Loops linearization to the graph resulting from the union. Dylan linearization [9] is a blend of both approaches. It takes successive minimals according to the union of the superclass linearizations. Hence, the linearization is monotonic too. When there is a choice, it uses the same criterion as Clos.

Besides the fact that these linearizations do not meet at least one requirement, their definitions and algorithms are more obscure and less intuitive for programmers. In contrast, C3 meets all requirements and its algorithm is much more intuitive, but its results are sometimes not easily predictable.

4.4. Prospects

Linearizations can be further improved in several ways.

Quasi-monotonicity. Monotonicity is sought but difficult to maintain. When defining a class C with two direct superclasses B_1 and B_2, a linearization conflict may occur, making it impossible to compute a monotonic linearization of C, regardless of the *lpo* between B_1 and B_2. However, the programmer may prefer a non-monotonic linearization to a failure. Hence, one can modify formula (20) as follows. In the alternative case, when, for all j, there is i such that $a_1^j \in (a_2^i, \ldots, a_{k_i}^i)$, j is then defined as the least integer such that a_1^j is minimal according to $\preceq$ among $\cup_i L_i$. Accordingly, $\forall i$, $L_i' = L_i \backslash \{a_1^j\}$.

The strategy proposed by Forman and Danforth [45] involves considering whether the disagreement induced by a linearization conflict is "serious". For that, their object model is parameterized by a few functions that allow the programmer to specify serious disagreements. For instance, the function could check that there is at least one method that is not monotonically linearized. If there is no local property conflict in the situation considered, the disagreement is not serious, since all linear extensions are equivalent. For his part, Ernst [40] considers linearizations as total preorders and proposes to unify all classes in a cycle by merging their definitions. Of course, such merging may induce some conflicts between local properties, but the author does not analyze them. On the contrary, in a subsequent paper, he seems to consider that the idea is not feasible [41].

Partial class linearizations. It follows from Definitions 4.1 and 4.2 that linearizations are only intended to order local properties so as to be able to select and combine them. Therefore, only $llin(g, c)$ should be required to be monotonic linear extensions. For instance, if there are no local property conflicts, local precedence orders do not matter, and any linear extension makes a good linearization. In the case of a linearization conflict, if there is no conflict between local properties in the cycle, this cycle in $clin(c)$ does not appear in any $llin(g, c)$, and the cycle ordering does not matter at all. Hence, instead of computing $clin(c)$ as a total order, it could be restricted to a partial order that would only totally order all local properties for each global property $g \in G_c$. More precisely, $\preceq_{clin(c)}$ could be defined as $\bigcup_{g \in G_c} def(\preceq_{llin(g,c)})$. Monotonicity should thus usually, but not always, be preserved.

Specific property linearizations. An alternative involves allowing the programmer to specify a partial linearization for each global property. Indeed, for two global properties, it is not necessary to combine their local properties in the same order. Each class could thus provide a default linearization which could be overridden by the programmer for individual global properties. This is, however, just a research issue. In contrast to the previous proposals which can remain hidden from the programmer, specific linearizations would need some syntactic constructs.

4.5. Conclusion on linearizations

Linearizations have often been criticized.

- As a selection mechanism, they would be arbitrary. However, as linear extensions, they represent only a default selection mechanism in case of conflict, and the programmer can always switch it off by redefining the considered property in the class introducing the conflict. Hence, the only assumption is the masking rule and, in the situation of Fig. 5b, programmers must redefine `area` in `Square` if they disagree with masking.
- As a combination mechanism, linearizations would break class modularity because $l \ll cnm(c, l)$ is not verified when there are conflicts [91]. This seems, however, unavoidable in method combination when inheritance is multiple, and modularity breaking is not greater than with usual late binding. Indeed, the essence of object-oriented programming is that a function call can invoke a statically unforeseeable function. Moreover, linearizations can coexist with `super`, but the supposed antimodular behavior of linearizations must be carefully balanced against double evaluations yielded by qualified `super`.
- The choice of a specific linearization would be arbitrary. Actually, all arguments lead us to think that C3 is the best known linearization. Moreover, according to [55], it should be the only one because it also satisfies the extended precedence order.
- Linearizations would be hard for programmers to understand. This has actually been improved. The declarative definition of C3 is far more comprehensive than the algorithmic definitions of Clos and Dylan linearizations, and its transitivity-free variant should avoid misunderstandings. Forman and Danforth [45] also published a pedagogical presentation of linearizations. Moreover, development tools like Eclipse could easily provide programmers with some help for solving conflicts, combining methods, proposing *lpo*, and giving precise diagnoses of non-monotonic situations. Actually, as for any high-level construct, the intuition about linearization is a matter of habit. Admittedly, all these algorithms, even C3, are somewhat unpredictable. However, it would seem that Clos and Python programmers have adopted linearizations without too much trouble.

5. Static typing

Static typing is generally beyond the scope of this paper. Hence, this section only examines how static typing interacts with multiple inheritance and how type systems could be adapted to the metamodel. A program fragment is said to be *type safe* if it does not cause type errors to occur at run time; otherwise it is *unsafe*. Usually, a type system is said to be safe if all unsafe fragments are rejected. Recently, Cardelli [17] proposed a slightly different definition, whereby only *untrapped* errors

are rejected. An error is trapped if the compiler detects its possibility at compile time and generates the code for checking it at run time. Hereafter, we shall speak of *strict* and *permissive* type safety policies. Hence, all unsafe fragments will be rejected in a strict policy, but some specific unsafe fragments can be accepted in a permissive policy.

5.1. Record types and global properties

Usual type theories apply to our framework, with just a slight difference. Object types are generally presented as *record types*, i.e. functions from *names* to *types* [16]. However, names are now inadequate, and they must be replaced by global properties. This should not have any effect on the considered type system, apart from making types and properties *class based*, in the sense of [81], i.e. each property is introduced by a single class. Therefore, a possible type system associated with the metamodel might be that of Cardelli [16], where each class has a record type that associates a type with each known global property. Substituting global properties for names would likely be sufficient to adapt the metamodel to any other formal semantics of object-oriented programs such as object calculi [1].

5.2. Specialization vs. subtyping

It is commonly agreed that classes are not types and specialization is not subtyping [26]. However, apart from higher-order types, commonly used languages like C++, C#, Eiffel and Java identify classes (and interfaces) to types and class specialization ($\preceq$) to subtyping ($<:$), with the proviso that class specialization is constrained to coincide with subtyping.[5] This constraint concerns property redefinition, through the well-known *contravariance rule*, which states that return types must be covariant but parameter types contravariant [16,21]. However, class specialization and static typing entail a much debated tradeoff between type safety and expressiveness. The contravariance rule is essential in a strict type safe policy, but some authors argue that strictly contravariant parameter types have turned out to be useless for modeling real-world applications. For instance, Shang [89] ends his often cited paper by "*We can live without contravariance*". On the contrary, modeling the real world often requires covariant parameter types. Shang [89], Meyer [72], Ducournau [29], Büttner and Gogolla [15] present arguments in favor of this covariant policy. In practice, most languages have type-invariant parameters and covariant return types, thus ensuring strict type safety. In contrast, some other languages, e.g. Eiffel, Beta, O_2 [7] and Java/C# (for arrays only), have adopted a covariant policy that ensures permissive type safety.

CWA versus OWA. The unsafety yielded by covariance is highly related to whether type checking is done under the *closed-world assumption* (CWA) or the *open-world assumption* (OWA). The CWA is a condition of the *system-level safety* argued by Meyer [72]. It is unfortunately contrary to the reusability and modularity principles. Under the OWA, covariance implies that all method calls are potentially unsafe, except in one of the following two specific situations: (i) all parameter types cannot have proper subtypes because they are primitive or declared with the `final` keyword (also known as `sealed` in C#); and (ii) the receiver is proven to always have an *exact type* [14], as in Eiffel *catcall rule*, i.e. its static and dynamic types are equal. Obviously, unconstrained covariance presents too much unsafety under the OWA, i.e. with a strict policy, most method calls would be rejected and, with a permissive policy, numerous compiler-generated run-time type checks would make the code inefficient. Therefore, a sensible tradeoff between expressivity, efficiency, and type safety would be to declare whether a global property is covariant or not when introducing it. This must be done, however, for each parameter whose type is not *final*.

5.3. Virtual types and intersection types

Virtual types represent a safer alternative to covariant parameters which can be strictly safe, according to Torgersen [98], or more permissive, as in Beta. They must not be confused with *virtual classes* [67], which are *virtual* (in the sense discussed on page 558) *inner classes*. We follow here the presentation of virtual types by Torgersen [98], while also considering the more permissive Beta subtyping rule.

Virtual types. A virtual type is a kind of attribute declared in a class and denoting a type. The metamodel applies to virtual types in a straightforward way. In some class C, the declaration of a virtual type T may take two forms, either $T <: D$ or $T = D$, where T is the name of the virtual type and D some "concrete" type, e.g. a class name. The former forces T to be a subtype of D in C and all subclasses of C, and the latter represents a *final* declaration that sets T for all subclasses, including C. If D is *final*, both forms are equivalent—hence, we assume that $T <: D$ implies that D is not *final*. The key idea of virtual types is that (i) as virtual types they must be covariantly redefined unless they are final ($T = D$), (ii) as a parameter or return type they are syntactically invariant, (iii) they have no proper subtypes unless they are final, and (iv) they are not even their own subtype unless they are final or dependent on the same receiver. Besides these explicit virtual types, `SelfType` (also known as `MyType` or `ThisType`) represents the implicit declaration `SelfType`$<:$ C in each class C. It corresponds to `like Current` in Eiffel and can be thought of as the static type which represents the dynamic type of `self`. `SelfType` is

[5] Like most assertions about C++, this one presents exceptions, e.g. *private inheritance*.

never final, unless the class itself is final. Generally speaking, we are not in favour of a `final` keyword forbidding further specialization, which would be against reusability. However, some final classes are natural, like primitive types, and final virtual types represent an acceptable compensation for type safety.

When a variable, say x, is typed by a virtual type T (denoted $x : T$), the dynamic type of the x value must be a subtype of the virtual type T in the *dynamic type* of `self`. In other words, $x : T$ stands for $x :$ `self`$.T$. Let us now consider the code fragment `{y:A; z:D; y.foo(z);}`, whereby the `foo` parameter has the virtual type T, and $T <: D$ in A. The call is unsafe because the dynamic type of `y` could be $B \prec A$, such that B redefines $T = E$, where E is some proper subclass of D. However, the permissive rule accepts it, at the expense of a compiler-inserted run-time type check.

Under the CWA, virtual types and covariant parameters are strictly equivalent—when a parameter type is redefined in a subclass, a virtual type is introduced in the superclass and substituted for the parameter type. Hence, under the OWA, virtual types are a convenient way to declare that a parameter type is covariant and, accordingly, all parameter types become syntactically invariant. If such a type redefinition has not been anticipated when introducing the global property, the programmer must manage the redefinition in an exceptional way, i.e. by explicitly checking the parameter dynamic type. This overall analysis is close to that of Shang [89].

Return types. Regarding the return type, free covariant redefinition is type safe. However, parameter and return types are often correlated, and there are strong arguments in favor of also keeping return types syntactically invariant. Let $C_k \prec C_{k-1} \prec \cdots \prec C_0$ and $D_k \prec D_{k-1} \prec \cdots \prec D_0$ be two specialization chains, and T be a virtual type introduced in C_0 and defined as $T <: D_i$ in each class C_i. Suppose also that `foo` and `bar` are methods introduced in C_0, the former with a parameter of type T and the latter with return type T. Then, if the static type of `x` is some C_i, `x.foo(x.bar())` is safe because it involves the *same virtual type*, depending on the *same receiver*. In contrast, if the return type of `bar` were D_i in each C_i, `x.foo(x.bar())` would be unsafe. We shall see that method combination leads to a similar conclusion. Hence, it might also be preferable to forbid explicit return type redefinition, and the method and attribute types would remain syntactically invariant, with all covariant redefinition going through virtual types.

Virtual type conflicts and class compatibility. So far, nothing is new apart from substituting global properties for names. Local property conflicts require combining types. It follows from the specification of virtual types that the conflict solution is quite constrained when the conflicting property represents a virtual type. Only some of them are erroneous.

Definition 5.1 (*Virtual Type Conflict Error*)**.** Let D be a class that presents a local property conflict (Definition 2.12) on a virtual type T, with the class conflict set $\{B, C\}$. Suppose that T is respectively defined, in B and C, in relation with the types t_B and t_C. This conflict is an error only in one of the following cases:

- $T = t_B$, $T = t_C$ and $t_B \neq t_C$;
- $T <: t_B$, $T = t_C$ and t_C is not a subtype of t_B;
- $T <: t_B$, $T <: t_C$, and t_B and t_C are *incompatible* (Definition 5.2).

In all three cases, the error cannot be overcome, i.e. the programmer cannot solve the conflict, and the compiler cannot compile D. Moreover, this gives a complete characterization of *class compatibility*.

Definition 5.2 (*Class Compatibility*)**.** Two $\preceq$-unrelated classes B and C are compatible, i.e. they can have common subclasses and instances, unless: (i) one of them is *final*, and hence cannot have proper subclasses, or (ii) they present an error in a virtual type conflict (Definition 5.1).

This possibly entails cyclic constraints such that B and C are compatible iff t_B and t_C are compatible. However, the rule is optimistic as compatibility is the default, and only incompatibility must be justified.

Compatibility can also be made a feature of the language. For instance, the Fortress language [2] allows the programmer to specify that a class (actually a *trait*; see Section 6.2) is incompatible with other classes. This interesting feature is close to the negation in *description logics* [5]. The programmer can thus express that `Man` and `Woman` are incompatible, hence forbidding the definition of `Hermaphrodite`. However, such incompatibility cannot be complete without the CWA.

Intersection types. The third conflict case in Definition 5.1 implies *meet types* [16] or *intersection types* [85,25], i.e. T must be a subtype of $t_B \cap t_C$. This amounts to solving the conflict by combining the conflicting values. There are essential differences with Cardelli's meet types [16]: (i) Cardelli's incompatibility is here impossible, since property names are replaced by global properties with statically invariant signatures; (ii) in contrast, the condition of compatibility is here recursively expanded to virtual types; and (iii) in the metamodel, types are class based, and hence a class implementing an intersection type is required. Finally, strict contravariance would require *union types*, and the only difference is that a union type always exists.

Class compatibility is, however, not exempt from the possibility of *actual type conflicts*, since $x \cap y$ is undefined when there are no subclasses common to x and y, at the time when a value of type $x \cap y$ is needed. However, this error cannot be detected under the OWA, since all future subclasses do not have to be known. A solution might be to consider that a class that yields a proper intersection type is abstract, forcing the programmer to define at least a subclass which would specify a concrete subtype in the intersection. This would be similar to the aforementioned solution to local property conflicts proposed in [77] (see Section 3.2.2). This is, however, questionable, and an argument against it is that a class is not declared abstract only

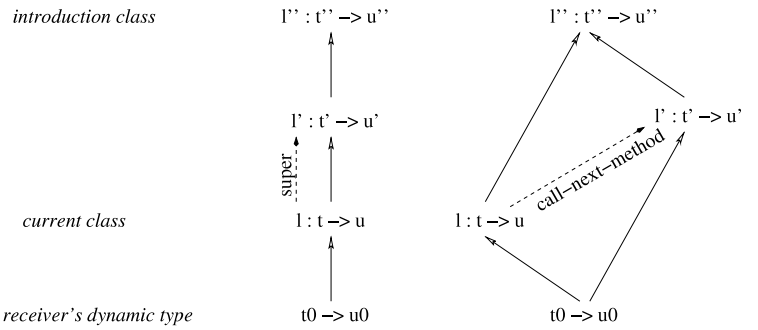


Fig. 6. Typing `super` and `call-next-method`. Solid lines represent redefinition and implied subtyping, and dashed lines calls to `super` and `call-next-method`.

because it involves some abstract class as a type. Suppose that C has a `foo` method with a parameter type D. The fact that D is abstract does not make C abstract. Hence, if D is replaced by a virtual type that involves an intersection type, this virtual type is somewhat abstract, but it is unclear whether it should make C abstract. Actually, the only conclusion that can be drawn is that `foo` is meaningless while the abstract class D, or the considered intersection type, has no concrete subclass, as if `foo` itself were abstract.

Finally, Definitions 5.1 and 5.2 easily generalize to conflict sets and compatibility of any cardinality, but actual run-time compatibility may require a class to be present at the intersection of more than two types.

5.4. Genericity

Virtual types also represent an alternative to parameterized classes [97,56]. A virtual type of the form $T <: D$ in the class A would correspond to the parameterized class $A\langle T <: D\rangle$, where the formal parameter T is bounded by D. A subclass A' of A fixing $T = D'$ would correspond to the instantiation $A\langle D'\rangle$. This is not an equivalence, just a strong similarity. This suggests that *formal type parameters* might be included in the metamodel in a way analogous to virtual types.

Accordingly, the formal type T in $A\langle T <: D\rangle$ corresponds to a new kind of property. This is an introduction if A is not declared to inherit another class, say $B\langle T\rangle$, parameterized by the same T; otherwise, it is a redefinition. Regarding properties and the metamodel, the only difference with usual properties is that the name of a formal type has lexical scope. Moreover, in case of multiple parameters, the correspondence between the parameter and the underlying global property depends on the parameter position.

A complete integration to the metamodel would require another association, namely the *instantiation* corresponding to substituting a concrete type for a formal type. We do not formalize it here, and focus on multiple inheritance. Regarding conflicts, the situation is quite simple. First, there are no global property conflicts, since there is no name, and positions are lexical. An original situation, unrelated to multiple inheritance, may occur when $A\langle T\rangle$ is declared to inherit $B\langle T, T\rangle$. This means that the two parameters of B are unified in A and must be instantiated in parallel in A subclasses. Local property conflicts are defined and managed in a way similar to virtual types. It does not make sense to define a common subclass of $A\langle C_1\rangle$ and $A\langle C_2\rangle$ when C_1 and C_2 are different concrete types. This is of course consistent with the formal view of parameterized types as functions from types to types.

5.5. Method combination

Method combination gives rise to some interesting typing questions. In practice, calls to `super` or `call-next-method` (Section 3.2.3) often represent one of two specific pipeline patterns: (i) the method parameters are passed to the callee as they were received by the caller; conversely, (ii) the value returned by the callee is returned at once by the caller (*tail call* in the following). In the former case, the parameter can be implicit, and both cases can be combined in a `return super` instruction without parameters. In Clos, the `call-next-method` syntax allows for the former case, as its usage without parameters implies passing all parameters as they are received. Both cases are of course type safe in a type-invariant framework, including virtual types. This means that, with the permissive rule, the primary method call may need run-time type checking, but all subsequent `call-next-method` with implied parameters or tail calls are type safe.

However, in a type-variant setting, method combination is more paradoxical. In contrast to `super`, `call-next-method` is not a static call, and the type of the method actually called may be as general as its type in the introduction class. In Fig. 6 (right), `call-next-method` in method l might call l', whose type is statically unknown and is only statically known to be bounded by the type of l''. Therefore, (i) if the return types are strictly covariant, tail calls are unsafe; this is another argument in favor of using virtual types for return types; (ii) if the parameter types are strictly contravariant, `call-next-method` without parameters is unsafe; and, paradoxically, (iii) if the parameter types are strictly covariant and `call-next-method` is always without parameters, it is safe since $t_0 <: t \cap t'$.

6. Comparison with other approaches to multiple inheritance

As mentioned in the introduction, there are no two languages that fully agree on their specifications of multiple inheritance. There are also no languages that fully meet our requirements. In this section, we first examine the specifications of multiple inheritance in some mainstream languages. We then present a family of language features that have be proposed as alternatives to full multiple inheritance. The section ends with a review of some related works and a brief discussion of implementation.

6.1. Inheritance and metamodels in some object-oriented languages

We review the most commonly used languages hereafter with two special points in mind. Does the language allow the programmer to express any legal model? Could it be adapted to our proposal while ensuring backward compatibility?

6.1.1. Dynamic typing

In the Smalltalk terminology, "method" and "method selector" respectively denote local and global properties. Nevertheless, selectors are simply reified as symbols, and there is no equivalent for attributes (*instance variables* in Smalltalk terminology). In Clos [92], "method" and "generic functions" stand for local and global properties. They are reified, but multiple dispatch changes the model as they do not belong to classes, and hence are not inherited in the usual meaning (Section 6.3). As for attributes (*slots* in Clos terminology), they are reified into two kinds of *slot description*, which can be *direct* or *effective*, but both must be understood as local properties [59]. Moreover, as already mentioned, dynamic typing cannot distinguish between two properties with the same name, whether they are methods, generic functions, or slots. However, if attribute access is *encapsulated*, i.e. reserved to `self`, as in Smalltalk, attributes can be considered as if they were in static typing. As it is also a sensible way to efficiently implement attributes, this restriction concerning dynamic typing concerns only methods and types. The lack of static types does not hinder a fully qualified syntax, but it makes it impossible to substitute instances of the metamodel for method names when parsing method calls (Requirement 2.1). In the example in Fig. 3, at run time, a call site `x.department` could alternatively reference `Researcher:department` or `Teacher:department` according to the dynamic type of `x`. Therefore, either Requirement 2.1 is dropped, but the call site becomes ambiguous, or the requirement is kept but both `department` are unified. In dynamic typing, the aforementioned naming convention is the only way to prevent conflicts. For instance, in the Clos *meta-object protocol*, all generic functions related to classes are prefixed by "`class-`", e.g. `class-name`, `class-slots`, `class-precedence-list`, etc. Despite this limitation, these languages are the only ones whose terminology is at least partly suitable for distinguishing between the two key notions that we have called local and global properties. In the following languages, one word (method, feature, etc.) stands for both notions.

6.1.2. Static typing

Multiple-subtyping languages. Java-like languages require a simple adaptation of the metamodel. Java class hierarchies are made of two kinds of entity, classes, and interfaces, which are disjoint except with respect to the hierarchy root. Let XC and XI, respectively, denote the sets of classes and interfaces; then $X = XC \cup XI$ and $XC \cap XI = \{\texttt{java.lang.Object}\}$. Indeed, `java.lang.Object` has two distinct roles, as a class (since it can be instantiated) and as an interface, since it is the type-system root, and interfaces can only extend interfaces. Classes are in single inheritance and $(XC, \prec_d)$ is a tree rooted in `java.lang.Object`. Interfaces imply multiple subtyping but can only specialize interfaces; hence $\prec$ is a subset of $(XC \times X) \cup (XI \times XI)$. Interfaces define only *abstract* methods (remember that static properties are not considered here). This type system can be understood as the result of statically typing Smalltalk, by adding interfaces for all methods introduced by more than one class. So far, Java is fully compatible with the metamodel, and C# too. A minor restriction with respect to static overloading has been lifted in Java 1.5 [3]. However, multiple inheritance is possible with interfaces and, when a global property conflict occurs, Java cannot distinguish between two methods with the same signature. Moreover, the compatibility of return types is reduced to strict equality. Accordingly, there is no reification of global properties, either in the introspection facilities (package `java.lang.reflect`), or in reflective extensions of the language, such as Open Java or Javassist, though Chiba [22] acknowledges the need for a metamodel. In contrast, a feature of C# called *explicit interface implementation* provides full qualification in a restricted way. When a class defines a method that has been introduced by several interfaces, the method name can be fully qualified. This allows the programmer to deal with global property conflicts in the right way, but the syntax can only be used at the definition level, and the ambiguity must be solved at method call with explicit upcasts. Moreover, the qualification is optional, even in a conflicting situation, and is restricted to directly implemented interfaces. This resembles a correction of the C++ manual renaming (Fig. 4, page 568), but the mixing with static overloading, through the possibly implied `new` keyword, makes the specifications confusing. In Java, a proposal for *hygienic methods* has been made, which involves full qualification by the introduction class or interface [64]. However, this qualification is uniform and the unqualified syntax is no longer possible.

```
class A
feature
      foo is do ... end
end

class B
inherit
      A rename foo as bar select bar end
      A rename foo as baz end
feature
      foo is do ... end
end
```

Class *B* inherits *A* by renaming `foo` as `bar`. The `select` keyword can be understood as if `A:foo` and `B:bar` were the same global property; This means that calling `foo` on a receiver typed by *A* while being an instance of *B* amounts to calling `bar`. Then, *B* inherits *A* again by renaming `foo` as `baz`. This roughly amounts to introducing a new global property `baz` which defaults to the inherited local property `A:foo`::A. Finally, *B* introduces a new global property `foo`, in a manner similar to the `reintroduce` and `new` keywords of Object Pascal and C#; `A:foo` and `B:foo` are both different and unrelated.

```
class A
feature
      foo is do ... end
end

class B
inherit
      A redefine foo select foo end
      A rename foo as foo_old end
feature
      foo is do ... foo_old ... end
end
```

Class *B* inherits *A* by first redefining and selecting `foo`. Then, *B* inherits *A* again by renaming `foo` as `foo_old`. Finally, *B* redefines `foo`, by calling `foo_old`. This is equivalent to calling `Precursor`. However, `foo_old` is now a newly introduced global property which can be called from anywhere and redefined, thus exceeding the specifications of call to `super`.

Both examples were tested under `ISE EiffelStudio version 6.4.7.9263 GPL Edition - linux-x86`.

Fig. 7. Repeated inheritance in Eiffel.

Eiffel. In Eiffel terminology, "feature" stands for property, without distinguishing between the two kinds, even though an informal notion of *feature seed* could be understood as the introduction of global properties [71]. Feature renaming allows the programmer to deal with global property conflicts in the desired way. However, a conservative use of renaming is advisable, since full usage of the `rename`, `select`, and `undefine` keywords is not compatible with the metamodel. Like C++ (see below), Eiffel provides repeated inheritance,[6] in diamond-like situations, but also with transitivity edges or even multiple inheritance of the same class. In all cases, this implies a lot of renamings. The difference with C++ is that repeated inheritance results here from explicit declarations, whereas it is implied by non-virtual inheritance in C++. In Fig. 7, class *B* inherits *A* twice in two different keyword combinations. The same results could be achieved with a diamond where *B* inherits *A* from two inheritance paths. Moreover, Eiffel does not follow the masking rule (Requirement 3.1) and finds extra local property conflicts, as in Fig. 5b. The programmer can, however, restore the desired behavior with the `undefine` keyword, but this is clumsy.

C++. Among the most commonly used languages, C++ is the least compatible with the metamodel. Besides a slight difference in static overloading which concerns visibility, multiple inheritance raises several difficulties. Our proposal corresponds to a systematic use of the `virtual`[7] keyword in inheritance. In C++ jargon, this is called *shared multiple inheritance*, instead of *repeated multiple inheritance* when `virtual` is omitted. Indeed, `virtual` avoids duplication of attributes introduced in a superclass inherited through multiple paths. Without `virtual`, in the diamond situation of Fig. 5, the attributes introduced by `Quadrilateral` would be duplicated in an instance of `Square`. This is often viewed as an acceptable alternative semantics, and it is thus important to explain why this language construct must actually be considered as "an abomination" [107]. Indeed, in the worst case, non-virtual inheritance can lead to a number of interpretations that are exponential in the class number. Consider, for instance, a chain of n diamonds made of classes A_i, B_i, C_i, and D_i, akin to Figs. 3a and 5a, such that $D_i = A_{i+1}$; then A_1 will be repeated 2^n times in the D_n layout. This exponential number of interpretations requires an equal number of syntactic expressions to be able to access each of them. Therefore, non-virtual inheritance should be used only on diamond-free hierarchies and, in the original paper describing it, Krogdahl [63] prohibits all diamond situations. Mixing virtual and non-virtual inheritance is also possible, though confusing. Overall, whereas virtual inheritance is fully reusable under the OWA (apart from global conflicts), non-virtual (or mixed virtual and non-virtual) inheritance cannot meet both requirements while avoiding diamonds. Furthermore, in C++, inheritance can be made private, thus changing the relationship between classes and types.

Anyway, when it is used in the most appropriate way for multiple inheritance, i.e. with public virtual inheritance, C++ does not allow the programmers to distinguish between two methods with the same signature introduced in different unrelated superclasses, and to fully redefine them. However, this does not apply to attributes whose definition complies with the metamodel when `virtual` is used. This asymmetry between methods and attributes represents one of the most striking flaws of the language specifications, e.g. two attributes named `foo` would be distinguished but their accessors `get_foo` would be confused with each other or would not be consistently redefinable in subclasses. Finally, in contrast to Eiffel, C++ signals local property conflicts according to the masking rule (Requirement 3.1).

Conclusions on mainstream languages. Overall, only Eiffel allows programmers to implement any multiple-inheritance model which complies with our metamodel. Conversely, not all Eiffel programs satisfy it. In a multiple subtyping setting,

6 In Eiffel terminology, this is called *replicated* inheritance and *repeated* inheritance denotes diamond-like situations.

7 Though more obscure, the use of `virtual` in inheritance is similar to its use for methods, i.e. a "virtual base class" (in C++ jargon) has a "redefinable" position in the object layout. See also page 558.

a similar conclusion holds for C#. Both languages are, however, rather confusing on these specific aspects. In contrast, C++ and Java do not allow programmers to define a class as a subclass of any two unrelated preexisting classes (or interfaces), while respecting the metamodel. When each of the two superclasses introduces a global property of the same name, it is impossible to express that their common subclass has both properties in a fully redefinable way. Moreover, if the return types are considered to be incompatible according to Definition 5.2 or some more restricted definition as in Java, the conflicting classes will be incompatible, and the subclass will not even compile.

At first sight, all three languages could be slightly modified to comply with our proposal at the global property level. Both Java and C++ require some fully qualified syntax, together with checking for ambiguous unqualified names. This should be sufficient for making these languages compliant with our metamodel. The question of backward compatibility could be addressed in the same way as in C#, whereby full qualification is possible but not mandatory. However, in C++, this adaptation only applies to public virtual inheritance. Ensuring backward compatibility for diamond-free non-virtual inheritance is certainly more complicated. In Eiffel, once a feature has been renamed, its old name should no longer be usable in the considered class for introducing a new global property (this resembles static overloading) or for another renaming.

6.1.3. Modeling languages

Eventually, metamodeling cannot be considered without also considering UML, where all entities are metadefined. In fact, concerning properties, the UML metamodel is left unfinished, i.e. the *Features diagram* of [80] shows only one kind of entity called "feature". On page 38 of [80], the specification states that "*one classifier* [namely, a class] *may specialize another by adding or redefining features*". There is no way to map this single term to our metamodel. Büttner and Gogolla [15] explain how specialization and redefinition are specified in UML 2.0. For instance, they make it clear how UML 2 introduces features like covariant redefinition and static overloading whose combination would lead to unspecified semantics. However, they do not consider virtual types, whereas covariance remains compatible with static overloading when covariant parameter types are restricted to virtual types [98].

6.2. Alternatives to full multiple inheritance—mixins and traits

Several restrictions of multiple inheritance have been proposed. Like multiple subtyping of Java interfaces, they involve distinguishing between first-class and second-class classes, and this may be coupled with a distinction between class specialization and subtyping. In all cases, the restriction of specialization to first-class classes is constrained to exclude diamonds. A variety of languages like Scala [78], Fortress [2] and Ruby [43] are based on the close notions of *mixins* [13] or *traits* [27]. A recent proposal distinguishes between `extends` and `requires` relationships [69], and seems to belong to the same family.

Mixins (also known as *mixin classes*) are commonly presented as an alternative to full multiple inheritance. They first appeared as a programming style in Lisp-based languages. The term was originally coined according to the Flavors metaphor that compares classes to ice-cream flavors. The hierarchy root is vanilla flavor, which is common to all ice creams, and inheritance is viewed as mixing in flavors [105]. This confusion between aggregation and specialization has unfortunately upset many authors. It can be found, for instance, in the pastry example of multiple inheritance in the Clos specifications [92], and it is likely the basis of C++ and Eiffel repeated inheritance. See, for instance, [18]. Later, the mixin notion becomes an explicit pattern or even a first-class entity in theories or actual programming languages. Mixin proposals are numerous and variable, and cannot all be covered in this section. Generally speaking, mixins are *abstract classes* with some restrictions in the way they are defined and related to other classes or mixins. Above all, a mixin is not self-sufficient as it must be used to qualify a class. To make a distinction from linguistics, a class is *categorematic*, like a noun, whereas a *mixin* is *syncategorematic*, like an adjective [65].

Traits are variants of mixins which provide a more formal way of combining them, with a finer grain. In the original proposal of Ducasse et al. [27], traits differ from classes and mixins by the fact they are not organized in a hierarchy. They must rather be explicitly combined with a kind of algebra to form classes. Different operators allow the programmer to precisely manage name conflicts, but at the local property level only. In a static typing framework, traits are often types [76], like usual mixins, but not always. However, the difference between both terms is often weaker, and their usage is not always consistent. For instance, Scala uses "trait" instead of "mixin", but the implied notion is markedly closer to what we call mixin here. "Trait" was previously used in Self [100] with yet another, though similar, meaning. In C++, "trait" denotes a programming pattern which allows the programmer to somewhat "refine" primitive types, especially characters. Original traits are far from our Aristotelian view of classes and inheritance; they must instead be considered as containers of code. Thus, in the following, we present a definition of mixins that is more class-like and close to that of traits in Scala.

Let XC and XM, respectively, denote the sets of classes and mixins, with $X = XC \uplus XM$. A class in XC or a mixin in XM can have a single direct superclass in XC, but any number of superclasses in XM, with the additional constraint that the restriction of $\prec$ to XC must form a tree. Therefore, one can define a class C such that C `extends` B `with` M, where B is the direct superclass of C and M is a *mixin*. However, the single direct superclass of M (in XC) must also be a superclass of B. Actually, the specifications are somewhat more general, and the single mixin M can be replaced by a set of $\prec$-related mixins such that all their superclasses in XC must be superclasses of B. Intuitively, the effect of this definition is to copy the

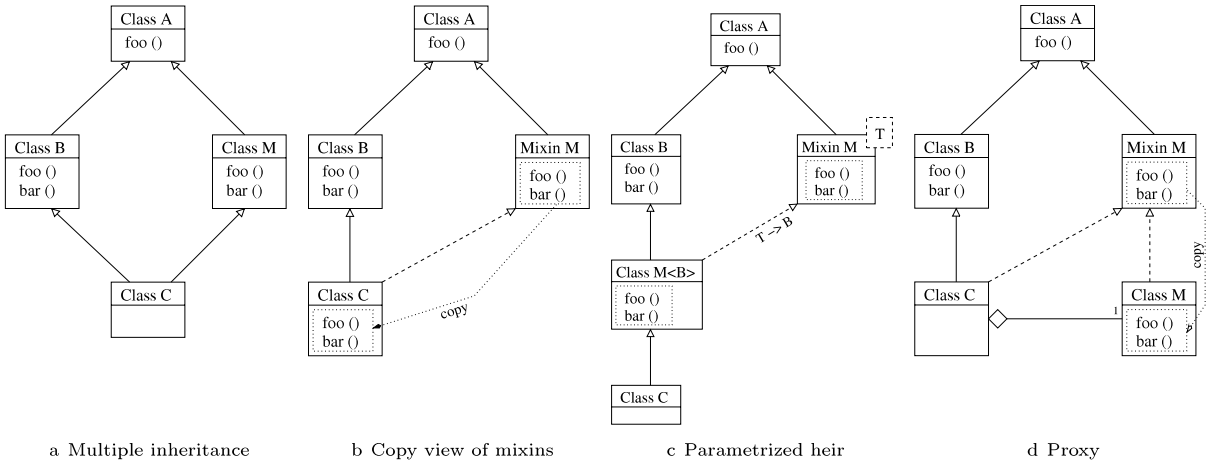


Fig. 8. Multiple inheritance and mixins – (a) in multiple inheritance, the example reproduces the two conflicting situations in Figs. 3 and 5 – `foo` stands for `area` and `bar` for `department`; (b) the same example involving a copy view of mixins; (c) mixins as parameterized heir classes; (d) mixins as proxies. Solid arrows denote class specialization (Java `extends`) and dashed arrows represent interface implementation (Java `implements`).

definition of *M* into *B* (Fig. 8b). An alternative and more formal view involves *parameterized heir classes*, whereby *C* `extends` *M*⟨*B*⟩ (Fig. 8c). This is the common way of using mixins with C++ *templates* [102,90]. The *heterogeneous* implementation of templates[8] makes it possible, but mixins cannot be separately compiled. In contrast, the *homogeneous* implementation of generics makes it impossible in Java. Finally, in a last view compatible with a *homogeneous* implementation, *M* is transformed into a *proxy*, which involves both an interface and a class: *C* implements the interface and is associated with the class by an aggregation, in such a way that each instance of *C* contains exactly one instance of *M* (Fig. 8d). The latter approach has also been called *automated delegation* [103]. The Scala implementation relies on a mix of copy and proxy. *M* is translated into an interface implemented by *C*, with the methods of *M* being compiled into Java static methods that are called from compiler-generated methods of *C*. The attributes of *M* are just copied into *C*.

Actually, mixins are not exempt from multiple inheritance conflicts; for example, *B* and *M* may each introduce a property with the same `bar` name, or redefine the same `foo` property introduced in their common superclass *A* (Fig. 8). Hence, mixins are not incompatible with the present metamodel, which could be extended to include them, in the same manner as for Java interfaces. Global property conflicts are exactly the same as in the general case, and they require the same solutions. Most mixin-based languages, e.g. Scala, do not recognize these conflicts, and solve them by *unification* (Section 3.1). MixJava is a notable exception, since it has a `view` keyword which resembles full qualification [44]. Regarding local property conflicts, there is no uniform policy among various mixin proposals, but all involve some explicit or implicit linearization. The previous definitions (by copy or parameterization) yield the same result, and the mixin *M* overrides the direct superclass *B*. Scala uses the same linearization as Common Loops where *M* (or the set of superclasses in *XM*) precedes the totally ordered superclasses (in *XC*).

6.3. Related features and works

Multiple dispatch. In its original specifications in Common Loops and Clos, *multiple dispatch* requires a deeply modified metamodel, since methods and *generic functions* are no longer defined in a class but instead across classes. In dynamic typing, this form of multiple dispatch reduces to single dispatch in the Cartesian powers of *X*, and local property conflicts are solved by linearizing these Cartesian powers. However, in a tuple of types, types are defined independently of each other, whereas a key feature of the metamodel and usual object-oriented programming is that classes are defined as a whole. Therefore, the reduction of multiple dispatch to single dispatch does not apply to the underlying metamodels.

In static typing, Castagna [21] proposed the notion of *overloaded functions*, which has almost nothing to do with *static overloading* and instead involves a form of multiple dispatch much more compatible with usual object-oriented programming. An overloaded function of type $\{t_i \rightarrow u_i\}_{i=1..n}$ has *n branches*, and a single branch is selected at run time as the most specific according to the dynamic type of its actual parameter. In the metamodel, a local property would be such an overloaded function and the integration is straightforward. Late binding is now a two-step process, as a kind of inverted version of static overloading: (i) a local property is first selected, as usual, according to the receiver dynamic type; and (ii) a branch is selected in the local property according to the dynamic types of all other parameters. Overloaded functions

[8] The implementations of generics lie between two extremes [79]. In heterogeneous implementation, e.g. C++ templates, each instance of the parameterized class is separately compiled. In homogeneous implementation, e.g. Java 1.5, a single instance is compiled, after replacing each formal type by its bound.

would thus seem to be a modular form of multiple dispatch that is compatible with the usual view of methods, since each overloaded function is defined as a whole in a class which is also defined as a whole. However, with multiple inheritance, branch selection introduces another kind of run-time conflict. Suppose that there are two branches $i \neq j$ in the considered overloaded function such that t_i and t_j have a common subclass, say C. Then the call is ambiguous when the actual parameter is an instance of C, unless there is another branch k such that $C <: t_k <: t_i, t_j$. Hence, Castagna adds a typing rule that requires such a branch t_k when C exists, but this rule implies the CWA. Under the OWA, the Fortress [2] solution involves defining a branch for each $t_i \cap t_j$ such that t_i and t_j are not incompatible. In spite of the language feature that allows the programmer to declare explicit incompatibility, it is potentially exponential. In contrast, MultiJava [23] disallows branch dispatching on interfaces, and a similar approach is taken in [69]. An alternative involves dropping the branch declarativity by ordering them. Hence, the actual type would become an ordered set $(t_i \rightarrow u_i)_{i=1..n}$, and the first branch that matches the actual parameter types is selected. This resembles the `typecase` construct, and it is likely the only way to maintain modularity with multiple inheritance. This ordering, however, seems to require the same properties as linearizations; namely, linear extensions would place the subtype before the supertypes, and monotonicity would ensure that the order specified in a class would be respected in the subclass.

On the distinction between homonymic properties. Nystrom et al. [77] analyze multiple inheritance in a way very similar to us, but due to the lack of metamodel the authors speak about *name conflicts* uniformly. Global property conflicts are solved by fully qualified syntax, whereas local property conflicts make the considered class *abstract*, and hence they require further redefinition. However, method combination is not addressed. The authors also claim that Borning and Ingalls [12] were the first to distinguish between properties according to the class which introduces them: "The distinction between name conflicts among methods introduced in a common base class and among methods introduced independently with possibly different semantics was made as early as 1982 by Borning and Ingalls". Actually, we carefully reread this four-page paper and did not find any trace of that. We only found a mention of *compound selectors* for disambiguating calls to `super`. Anyway, this 1982 paper is quite informal, and cannot yield precise semantics. For instance, it is impossible to determine whether the paper implies the *masking rule*, or not. Nystrom et al. also consider that their view of multiple inheritance derives from *intersection types* [25,85] (see Section 5). In the papers on intersection types referenced, we did not find any information about the way name conflicts are managed. It would seem that the intersection type just takes the union of inherited property names irrespective of introduction classes. In these affiliation proceedings, the historical paper of Cardelli [16] must also be ruled out. It proposes a theory of record types without classes, and hence without any notion of introduction, and it could have been more appropriately entitled "a semantics of multiple subtyping". Knudsen [61] attempts to analyze *name collisions* in multiple specialization hierarchies. Though he considers qualifying property names by some class, or even a class path, he does not consider the introduction class. Moreover, he concludes that the programmer is in charge of resolving all collisions, including redefinitions, whereas our proposal restricts the programmer's role to local property conflicts.

Altogether, to our knowledge [36], this idea of distinguishing between properties according to the class which introduces them goes back to some studies on the notion of *point of view*, in a dynamic typing and knowledge representation framework [20,39]. We are not aware of any prior mention of it. This idea was recently proposed in different ways, in C#, and with the aforementioned hygienic methods [64] and multiple nested inheritance [77]. The `view` keyword of MixJava [44] seems to belong to the same family. The idea has thus certainly ripened.

Higher-order metamodeling. Since Beta *virtual classes* [67], there have been many proposals of nested classes [42,77], where each class can in turn play the role of an ordinary class and/or the role of a *module* (or *package*) containing a set of classes. Ducournau et al. [37] propose a two-level metamodel, where the class–module metamodel is isomorphic to the property–class metamodel presented here. The *import* relationship between modules is then analogous to inheritance between classes and property redefinition is translated, at the class-module level, by the *class refinement* notion which allows the programmer to incrementally define classes. Import conflicts are managed as inheritance conflicts, at both global and local levels.

6.4. Implementation

Implementation of multiple inheritance is beyond the scope of this paper. Interested readers are referred to [31] for a general survey and abstract assessment of object-oriented implementation techniques. Systematic experiments are reported in [38]. Multiple inheritance presents a noteworthy tradeoff between modularity, which relies on the OWA, and efficiency, which comes from the CWA. The overall conclusions about implementation would be that there are state-of-the-art implementations of multiple inheritance under the CWA, either at compile or link time; these implementations are as efficient as single inheritance implementations. Moreover, implementing multiple subtyping or mixins is not easier or more efficient than plain multiple inheritance, apart from attributes which remain in single inheritance. The observed efficiency of Java and .Net systems mostly comes from their *adaptive compilers* [4] which rely on temporary CWA and possible load-time recompilations. Finally, under the OWA, efficient implementations compatible with dynamic loading remain an open issue: the apparent scalability of C++ subobject-based implementation is likely due to intensive use of non-virtual inheritance, and that of Java interfaces is, in practice, doubtful. Although there are alternatives such as perfect hashing [30], we do not know any OWA implementation which does not entail significant overhead, *even when multiple inheritance is not used*.

7. Conclusion and perspectives

In this paper, we have proposed a metamodel of classes and properties which aims to give an intuitively sound and simple semantics to object-oriented programming and multiple inheritance. Besides language specifications, this metamodel can have several usages. It can serve as a conceptual support for programmers who want to map an object-oriented model to their Aristotelian intuition. It can (and of course should) serve as a basis for implementing all tools which are required for making object-oriented languages operational, namely compilers, run-time systems, development tools, meta-object protocols, etc. Here we have explored its consequences on the interpretation of multiple inheritance. The first and fundamental consequence is to distinguish between two different conceptual levels for properties, inheritance, and associated conflicts. The benefits of this analysis also apply to virtual types and genericity.

Global properties are intended to model the idea of the "same property" shared by different related classes. They represent the essence of properties. Global property conflicts are a pure matter of names—they can be easily and safely solved in any language, provided that this language offers some fully qualified syntax. However, in dynamic typing, this does not fully apply, and a naming convention is required for methods. *Local properties* model the implementation of a given global property in different classes. They may have conflicts when a class cannot choose between different implementations for the global property considered. This kind of conflict is quite different, since the choice requires the programmer's decision. Programming languages can only provide some constructs to help programmers in their choice. This distinction also sheds some light on the distinction between *static overloading* and *overriding* (called here *redefinition*). "Overloading" is often used for both, and this is confusing (not to say overloaded). *Static overloading* concerns different global properties, whereas *overriding* is a relationship between local properties of the same global property.

The proposed metamodel should be considered as a possible extension of the UML metamodel, as it makes the meaning of such fundamental notions more precise. The specifications of most languages could also adopt it while only marginally changing language syntax, programming habits, or program behavior. This would provide a terminological and conceptual basis for object-oriented programming, for either program documentation or teaching, and a sound basis for all tools which are dedicated to program manipulation—compilers, programming environment, etc. Our claim is that languages like Java, C++, and Eiffel would be markedly improved if they complied with this metamodel. In practice, at least for C++, this is however a dream, as such a compliance requires numerous changes which are incompatible with existing programs, even though the incompatible cases are likely quite marginal. Global property conflicts are not an obstacle to multiple inheritance. Their solution is syntactically simple and intuitive for all programmers. This is, however, a major flaw of both C++ and Java, which hinders reusability. A C#-like approach could apply to Java with full backward compatibility. Local property conflicts imply a remaining issue, i.e. method combination. There are several solutions (static calls, linearizations, mixins), but none of them is perfect. This is likely inherent to multiple inheritance. By the way, a last contribution concerns linearizations. They are widely used in dynamically typed languages, but our analysis shows that they could be used in static typing too, thus providing a practical solution to method combination. Moreover, we sketched several extensions of the linearization framework. First, monotonicity can be relaxed with the quasi-monotonic variant of C3. Partial and specific linearizations should also increase linearization flexibility. However, they require precise specifications and syntactic constructs that have yet to be studied.

In contrast to our proposal, mixins do not answer the whole question, and they need to be supplemented by our metamodel at the global property level, e.g. with full qualification. Indeed, the metamodel can be easily adapted, while preserving its principle, to include interfaces and mixins. At the local property level, mixins and linearizations are roughly equivalent. Mixins are sometimes defined by the fact that they are linearized, e.g. "*They were defined as classes that allow their superclass to be determined by linearization of multiple inheritance*" [90]. This condition is necessary but not sufficient, as Bracha and Cook [13] made it clear that mixins are not ordinary classes, whereas linearizations apply to unrestricted class hierarchies. Indeed, mixins are equivalent to a subset of linearization-based multiple inheritance hierarchies. Hence, linearization-based multiple inheritance permits a full mixin-like programming style, but mixins only permit a very restrictive use of multiple inheritance. Our view of multiple inheritance is both set theoretical, i.e. based on union of property sets, and order theoretical, i.e. based on partial and total orders. Multiple inheritance mostly amounts to combining, i.e. "mixing in", superclasses, by union of property sets, and ordering conflicting local properties, by linearization. Hence, "mixin" is often used in a loose way, and multiple inheritance and "mixins" are hard to distinguish. For instance, in [40], it is unclear whether "mixin" could be uniformly replaced by "class" without any change. A similar question applies to Scala. It would seem that the distinction between traits and classes results only, in this language, from the targeted run-time systems and libraries, namely Java or .Net, that provide only multiple subtyping. If a full multiple-inheritance run-time system had been available when the language was designed, this distinction would not have been necessary, and the language might have been specified with a single kind of class-like entity.[9] Finally, mixins have the disadvantage of adding a new kind of entity (mixins or traits) which are akin to classes, but different, rather ad hoc and which do not appear to be stable enough. In contrast, full multiple inheritance relies only on the simple notion of class which is conceptually well understood, after centuries of Aristotelian tradition.

[9] This is our interpretation of a discussion with Martin Odersky during his invited talk at the *Languages et Modèles à Objets* French-speaking Conference, in Nancy (France) in 2009.

To sum up, our conclusions on multiple inheritance are as follows.

1. There are state-of-the-art solutions for syntactic conflicts. The flaws of current production languages should be fixed, though backward compatibility likely makes this impossible in some languages. New languages should adopt sound specifications, namely fully qualified names, possibly with lexical-scope renaming. The C# language specifications and the proposal for *hygienic* methods in Java represent first steps in this direction. An Eiffel-like renaming remains also acceptable, in a restricted form.
2. Semantics conflicts are precisely defined; forcing redefinition of the conflicting property or making the class abstract is a sensible answer; linearization-based selection is an alternative.
3. Method combination remains the most controversial point. The supposedly antimodular behavior of linearizations must be balanced against the risk of multiple evaluations entailed by static calls.
4. Anyway, C3 is the right linearization, possibly in its quasi-monotonic variant. Partial or specific linearizations would improve the use of linearizations, and this should be the focus of further research. However, like any non-trivial algorithm, C3 can produce unexpected results.
5. Virtual types and invariant signatures seem to present the best tradeoff between expressiveness, efficiency, and type safety, especially in the context of multiple inheritance and method combination. Moreover, they yield a simple characterization of incompatible classes (Definition 5.2). Static overloading remains possible, though we do not defend it.
6. Development tools should implement the metamodel in order to efficiently support the programmer for multiple inheritance. Simple navigation in the program model would be straightforward, but intelligent help for linearizations needs further studies.
7. There are efficient ways of implementing multiple inheritance, but this remains partly an open issue under the OWA.

All of these conclusions hold in a static typing setting. Dynamic typing changes both the solution of syntactic conflicts and the basic implementation of object-oriented mechanisms. This will be the focus of future work.

References

[1] M. Abadi, L. Cardelli, A Theory of Objects, in: Monographs in Computer Science, Springer Verlag, 1996.
[2] E. Allen, D. Chase, J. Hallett, V. Luchanco, J.-W. Maessen, S. Ryu, G.L. Steele, S. Tobin-Hochstadt, The Fortress Language Specification, Version 1.0. Tech. Rep., Sun Microsystems, Inc., 2008.
[3] D. Ancona, S. Drossopoulou, E. Zucca, Overloading and inheritance, in: 8th Intl. Workshop on Foundations of Object-Oriented Languages, FOOL8, 2001.
[4] M. Arnold, S. Fink, D. Grove, M. Hind, P. Sweeney, A survey of adaptive optimization in virtual machines, Proceedings of the IEEE 93 (2) (2005) 449–466.
[5] F. Baader, D. McGuinness, D. Nardi, P. Patel-Schneider, The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, 2003.
[6] H. Baker, CLOStrophobia: its etiology and treatment, ACM OOPS Messenger 2 (4) (1991) 4–15.
[7] F. Bancilhon, C. Delobel, P. Kanellakis (Eds.), Building an Object-oriented Database System: The Story of 02, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
[8] J. Barnes, Programming In Ada 95, first ed., Addison-Wesley, 1995.
[9] K. Barrett, B. Cassels, P. Haahr, D.A. Moon, K. Playford, A.L.M. Shalit, P.T. Withington, A monotonic superclass linearization for Dylan, in: Proc. OOPSLA'96, SIGPLAN Notices 31 (10) (1996) 69–82. ACM.
[10] G. Birtwistle, O. Dahl, B. Myraug, K. Nygaard, Simula Begin, Petrocelli Charter, 1973.
[11] D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel, CommonLoops: merging Lisp and object-oriented programming, in: Proc. OOPSLA'86, SIGPLAN Notices 21 (11) (1986) 17–29. ACM.
[12] A. Borning, D. Ingalls, Multiple inheritance in Smalltalk-80, in: Proc. AAAI'82, 1982, pp. 234–237.
[13] G. Bracha, W. Cook, Mixin-based inheritance, in: Proc. OOPSLA/ECOOP'90, SIGPLAN Notices 25 (10) (1990) 303–311. ACM.
[14] K.B. Bruce, M. Odersky, P. Wadler, A statically safe alternative to virtual types, in: E. Jul (Ed.), Proc. ECOOP'98, in: LNCS, vol. 1445, Springer, 1998, pp. 523–549.
[15] F. Büttner, M. Gogolla, On generalization and overriding in UML 2.0, in: O. Patrascoiu (Ed.), OCL and Model Driven Engineering, UML 2004 Conf. Workshop, 2004, pp. 1–15.
[16] L. Cardelli, A semantics of multiple inheritance, Information and Computation 76 (1988) 138–164.
[17] L. Cardelli, Type systems, in: A.B. Tucker (Ed.), The Computer Science and Engineering Handbook, 2nd ed., CRC Press, 2004 (Chapter 97).
[18] T. Cargill, Aggregation: inheritance vs member objects, C++ Report 5 (1) (1993) 19–23.
[19] T.A. Cargill, Controversy: the case against multiple inheritance in C++, Computing Systems 4 (1) (1991) 69–82.
[20] B. Carré, J.-M. Geib, The point of view notion for multiple inheritance, in: Proc. OOPSLA/ECOOP'90, SIGPLAN Notices 25 (10) (1990) 312–321. ACM.
[21] G. Castagna, Object-oriented Programming: A Unified Foundation, Birkhaüser, 1997.
[22] S. Chiba, Javassist—a reflection-based programming wizard for Java, in: Proc. ACM OOPSLA Workshop on Reflective Programming in C++ and Java, 1998.
[23] C. Clifton, T. Millstein, G.T. Leavens, C. Chambers, Multijava: design rationale, compiler implementation, and applications, ACM Transactions on Programming Languages and Systems 28 (3) (2006) 517–575.
[24] P. Cointe, Metaclasses are first class: the ObjVlisp model, in: Proc. OOPSLA'87, SIGPLAN Notices 22 (12) (1987) 156–167. ACM.
[25] A.B. Compagnoni, B.C. Pierce, Higher order intersection types and multiple inheritance, Mathematical Structures in Computer Science 6 (5) (1996) 469–501.
[26] W. Cook, W. Hill, P. Canning, Inheritance is not subtyping, in: Proc. POPL'90, ACM, 1990, pp. 125–135.
[27] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. Black, Traits: a mechanism for fine-grained reuse, ACM Transactions on Programming Languages and Systems 28 (2) (2005) 331–388.
[28] R. Ducournau, Yet Another Frame-based Object-Oriented Language: YAFOOL Reference Manual, Sema Group, Montrouge, France, 1991.
[29] R. Ducournau, "Real World" as an argument for covariant specialization in programming and modeling, in: J.-M. Bruel, Z. Bellahsène (Eds.), Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc., in: LNCS, vol. 2426, Springer, 2002, pp. 3–12.
[30] R. Ducournau, Perfect hashing as an almost perfect subtype test, ACM Transactions on Programming Languages and Systems 30 (6) (2008) 1–56.
[31] R. Ducournau, Implementing statically typed object-oriented programming languages, ACM Comp. Surv. 43 (4) (2011).

[32] R. Ducournau, M. Habib, On some algorithms for multiple inheritance, in: J. Bezivin, P. Cointe, J.-M. Hullot, H. Liebermann (Eds.), Proc. ECOOP'87, in: LNCS, vol. 276, Springer, 1987, pp. 243–252.
[33] R. Ducournau, M. Habib, Masking and conflicts, or to inherit is not to own, in: M. Lenzerini, D. Nardi, M. Simi (Eds.), Inheritance Hierarchies in Knowledge Representation and Programming Languages, John Wiley & Sons, 1991, pp. 223–244 (Chapter 14).
[34] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, Monotonic conflict resolution mechanisms for inheritance, in: Proc. OOPSLA'92, SIGPLAN Notices 27 (10) (1992) 16–24. ACM.
[35] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, Proposal for a monotonic multiple inheritance linearization, in: Proc. OOPSLA'94, SIGPLAN Notices 29 (10) (1994) 164–175. ACM.
[36] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, A. Napoli, Le point sur l'héritage multiple, Technique et Science Informatiques 14 (3) (1995) 309–345.
[37] R. Ducournau, F. Morandat, J. Privat, Modules and class refinement: a metamodeling approach to object-oriented languages, Tech. Rep. LIRMM-07021, Université Montpellier 2, 2007.
[38] R. Ducournau, F. Morandat, J. Privat, Empirical assessment of object-oriented implementations with multiple inheritance and static typing, in: G.T. Leavens (Ed.), Proc. OOPSLA'09, SIGPLAN Notices 44 (10) (2009) 41–60. ACM.
[39] P. Dugerdil, Inheritance mechanism in the Objlog language: multiple selective and multiple vertical with points of view, in: M. Lenzerini, D. Nardi, M. Simi (Eds.), Inheritance Hierarchies in Knowledge Representation and Programming Languages, John Wiley & Sons, 1991, pp. 245–256 (Chapter 15).
[40] E. Ernst, Propagating class and method combination, in: R. Guerraoui (Ed.), Proc. ECOOP'99, in: LNCS, vol. 1628, Springer, 1999, pp. 67–91.
[41] E. Ernst, Safe dynamic multiple inheritance, Nordic Journal of Computing 9 (1) (2002) 191–208.
[42] E. Ernst, Higher-order hierarchies, in: L. Cardelli (Ed.), Proc. ECOOP'2003, in: LNCS, vol. 2743, Springer, 2003, pp. 303–329.
[43] D. Flanagan, Y. Matsumoto, The Ruby Programming Language, O'Reilly, 2008.
[44] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and mixins, in: Proc. POPL'98, ACM Press, 1998, pp. 171–183.
[45] I.R. Forman, S.H. Danforth, Putting Metaclasses to Work, Addison-Wesley, 1999.
[46] E. Gamma, K. Beck, Contributing to Eclipse: Principles, Patterns, and Plugins, Addison Wesley, 2003.
[47] A. Goldberg, D. Robson, Smalltalk: The Language and its Implementation, Addison-Wesley, 1983.
[48] D.S. Goldberg, R.B. Findler, M. Flatt, Super and inner: together at last!, in: J.M. Vlissides, D.C. Schmidt (Eds.), Proc. OOPSLA'04, SIGPLAN Notices 39 (10) (2004), 116–129. ACM.
[49] M. Grand, JAVA Language Reference, O'Reilly, 1997.
[50] S.P. Harbinson, Modula-3, Prentice Hall, 1992.
[51] J. Hickey, Introduction to the Objective Caml programming language, 2006. http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf.
[52] J. Horty, Some direct theories of nonmonotonic inheritance, in: D. Gabbay, C. Hogger (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming, in: Nonmonotonic Reasoning, vol. 2, Oxford University Press, 1994.
[53] D. Hovemeyer, W. Pugh, Finding bugs is easy, SIGPLAN Notices 39 (12) (2004) 92–106.
[54] M. Huchard, Another problematic multiple inheritance mechanism: Construction and destruction in C++ in the virtual inheritance case, J. Obj. Orient. Program. 13 (4) (2000) 6–12.
[55] M. Huchard, M.-L. Mugnier, M. Habib, R. Ducournau, Towards a unique multiple inheritance linearization, in: Mrazik, A. (Ed.), Proc. EurOOP'91, Sep. 1991.
[56] A. Igarashi, B.C. Pierce, Foundations for virtual types, in: R. Guerraoui (Ed.), Proc. ECOOP'99, in: LNCS, vol. 1628, Springer, 1999, pp. 161–185.
[57] ILOG, 1996. Power Classes reference manual, Version 1.4. ILOG, Gentilly.
[58] ISO/IEC 14882:2003, 2003. Programming languages — C++, International Organization for Standardization, Geneva, Switzerland.
[59] G. Kiczales, J. des Rivières, D. Bobrow, The Art of the Meta-Object Protocol, MIT Press, 1991.
[60] S.C. Kleene, Mathematical Logic, John Wiley, 1967.
[61] J.L. Knudsen, Name collision in multiple inheritance hierarchies, in: S. Gjessing, K. Nygaard (Eds.), Proc. ECOOP'88, in: LNCS, vol. 322, Springer, 1988, pp. 93–109.
[62] D.E. Knuth, The art of computer programming, in: Sorting and Searching, vol. 3, Addison-Wesley, 1973.
[63] S. Krogdahl, Multiple inheritance in Simula-like languages, BIT 25 (2) (1985) 318–326.
[64] J. Kusmierek, V. Bono, Hygienic methods — introducing HygJava, in: TOOLS 2007, in: JOT, vol. 10, ETH Zurich, 2007.
[65] A. Lalande, Vocabulaire technique et critique de la philosophie, Presses Universitaires de France, 1926.
[66] B.H. Liskov, J.M. Wing, A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems 16 (6) (1994) 1811–1841.
[67] O.L. Madsen, B. Møller-Pedersen, Virtual classes. A powerful mechanism in object-oriented programming, in: Proc. OOPSLA'89, SIGPLAN Notices 24 (10) (1989) 397–406. ACM.
[68] O.L. Madsen, B. Møller-Pedersen, K. Nygaard, Object-Oriented Programming in the Beta Programming Language, Addison-Wesley, 1993.
[69] D. Malayeri, J. Aldrich, CZ: multiple inheritance without diamonds, 2009, pp. 21–40.
[70] G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre, Object-Oriented Languages, Academic Press, London, 1991.
[71] B. Meyer, Eiffel: The Language, Prentice-Hall, 1992.
[72] B. Meyer, Object-Oriented Software Construction, 2nd ed., Prentice-Hall, 1997.
[73] Microsoft, C# Language specifications, v0.28. Technical Report, Microsoft Corporation, 2001.
[74] D. Moon, Object-oriented programming with Flavors, in: Proc. OOPSLA'86, SIGPLAN Notices 21 (11) (1986) 1–8. ACM.
[75] H. Mössenböck, Object-Oriented Programming in Oberon-2, Springer, 1993.
[76] O. Nierstrasz, S. Ducasse, N. Schärli, Flattening traits, Journal of Object Technology 5 (4) (2006) 129–146.
[77] N. Nystrom, X. Qi, A.C. Myers, $\mathcal{JE}$: nested intersection for scalable software composition, in: P.L. Tarr, W.R. Cook (Eds.), Proc. OOPSLA'06, SIGPLAN Notices 41 (10) (2006), 21–35. ACM.
[78] M. Odersky, L. Spoon, B. Venners, Programming in Scala, A Comprehensive Step-by-step Guide, Artima, 2008.
[79] M. Odersky, P. Wadler, Pizza into Java: translating theory into practice, in: Proc. POPL'97, ACM, 1997, pp. 146–159.
[80] OMG, Unified Modeling Language 2.0 superstructure specification, Technical Report, Object Management Group, 2004.
[81] W. Pugh, G. Weddell, Two-directional record layout for multiple inheritance, in: Proc. PLDI'90, SIGPLAN Notices 25 (6) (1990) 85–91. ACM.
[82] D. Rayside, G. Campbell, An Aristotelian introduction to classification, in: M. Huchard, R. Godin, A. Napoli (Eds.), ECOOP'2000 Workshop on Objects and Classification, A Natural Convergence. RR LIRMM 2000-095, 2000.
[83] D. Rayside, G. Campbell, An Aristotelian understanding of object-oriented programming, in: Proc. OOPSLA'00, SIGPLAN Notices 35 (10) (2000) 337–353. ACM.
[84] D. Rayside, K. Kontogiannis, On the syllogistic structure of object-oriented programming, in: Proc. of ICSE'01, 2001 pp. 113–122.
[85] J. Reynolds, Design of the programming language Forsythe, in: P. O'Hearn, R. Tennent (Eds.), Algol-like Languages, Birkhaüser, 1996.
[86] M. Sakkinen, A critique of the inheritance principles of C++, Computing Systems 5 (1) (1992) 69–110.
[87] A. Shalit, The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language, Addison-Wesley, 1997.
[88] Y.-P. Shan, T. Cargill, B. Cox, W. Cook, M. Loomis, A. Snyder, Is multiple inheritance essential to OOP? (panel), in: Proc. OOPSLA'93, SIGPLAN Notices 28 (10) (1993) 360–363. ACM.
[89] D.L. Shang, Are cows animals? 1996. http://www.visviva.com/transframe/papers/covar.htm.
[90] Y. Smaragdakis, D. Batory, Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 215–255.

[91] J. Snyder, Inheritance in object-oriented programming, in: M. Lenzerini, D. Nardi, M. Simi (Eds.), Inheritance Hierarchies in Knowledge Representation and Programming Languages, John Wiley & Sons, 1991, pp. 153–171 (Chapter 10).
[92] G.L. Steele, Common Lisp, The Language, 2nd ed., Digital Press, 1990.
[93] M. Stefik, D. Bobrow, Object-oriented programming: themes and variations, AI Magazine 6 (4) (1986) 40–62.
[94] B. Stroustrup, The C++ Programming Language, Special ed., Addison-Wesley, 2000.
[95] S.T. Taft, R.A. Duff, R.L. Brukardt, E. Ploedereder, P. Leroy (Eds.), Ada 2005 Reference Manual: Language and Standard Libraries, in: LNCS, vol. 4348, Springer, 2006.
[96] A. Taivalsaari, On the notion of inheritance, ACM Computing Surveys 28 (3) (1996) 438–479.
[97] K. Thorup, M. Torgersen, Unifying genericity: combining the benefits of virtual types and parameterized classes, in: R. Guerraoui (Ed.), Proc. ECOOP'99, in: LNCS, vol. 1628, Springer, 1999, pp. 186–204.
[98] M. Torgersen, Virtual types are statically safe, in: Elec. Proc. of the 5th Workshop on Foundations of Object-Oriented Languages, FOOL 5, 1998.
[99] D. Touretzky, The Mathematics of Inheritance, Morgan Kaufmann Publishers, Los Altos (CA), USA, 1986.
[100] D. Ungar, C. Chambers, B. Chang, U. Hølzle, Organizing programs without classes, Lisp and Symbolic Computation 4 (3) (1991) 223–242.
[101] G. van Rossum, F.L. Drake Jr., The Python Language Reference Manual, Network Theory Ltd., Sep. 2003.
[102] M. VanHilst, D. Notkin, Using role components to implement collaboration-based designs, in: Proc. OOPSLA'96, SIGPLAN Notices 31 (10) (1996) 359–369. ACM.
[103] J. Viega, B. Tutt, R. Behrends, Automated delegation is a viable alternative to multiple inheritance in class based languages, Tech. Rep. CS-98-03, University of Virginia, Charlottesville, VA, USA, 1998.
[104] J. Waldo, Controversy: the case for multiple inheritance in C++, Computing Systems 4 (2) (1991) 157–171.
[105] D. Weinreb, D. Moon, Flavors: message passing in the Lisp Machine, Tech. Rep. 602, MIT AI Lab, 1980.
[106] N. Wirth, The programming language Oberon, Software Practice & Experience 18 (7) (1988) 671–690.
[107] Y. Zibin, J. Gil, Incremental algorithms for dispatching in dynamically typed languages, in: Proc. POPL'03, SIGPLAN Notices 38 (1) (2003) 126–138. ACM.